Dynamic Reconfiguration of Business Processes

Leandro Nahabedian[1], Victor Braberman[1], Nicolás D'ippolito[1], Jeff Kramer[2], and Sebastián Uchitel[1,2]

[1] Universidad de Buenos Aires / CONICET, Argentina
`{lnahabedian | vbraber | ndippolito | suchitel}@dc.uba.ar`
[2] Department of Computing, Imperial College, London, UK
`j.kramer@imperial.ac.uk`

Abstract. Organisations require that their business processes reflect their evolving practices by maintaining compliance with their policies, strategies and regulations. Designing workflows which satisfy these requirements is complex and error-prone. Business process reconfiguration is even more challenging as not only a new workflow must be devised but also an understanding of how the transition between the old and new workflow must be managed. Transition requirements can include both domain independent, such as delayed and immediate change, or user-defined domain specific requirements. In this paper we present a fully automated technique which uses control synthesis to not only produce correct-by-construction workflows from business process requirements but also to compute a reconfiguration process that guarantees the evolution from an old workflow to a new one while satisfying any user-defined transition requirements. The approach is validated using three examples from the BPM Academic Initiative described as Dynamic Condition Response Graphs which we reconfigured for a variety of transitions requirements.

Keywords: Dynamic Reconfiguration, Controller Synthesis, DCR Graph

1 Introduction

Business processes are invaluable for ensuring that task and activity execution achieves business objectives. Workflows, operational representations of business processes, are typically derived from requirements in a manual process that is complex and error-prone. Organisations require that their business processes reflect their evolving practices maintaining compliance with their policies, strategies and regulations (e.g., [27]). Workflows must be evolved accordingly too. *Business process reconfiguration* involves not only devising the new workflow but also dynamically changing the old workflow with the new one.

Key to reconfiguration is understanding how the transition between the old and new workflow should be managed. Domain independent transition requirements have been studied extensively. For instance, [9] discusses "immediate" reconfiguration requirements that assert 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. “Delayed” reconfiguration asserts that living instances must finish using the old workflow, while fresh instances are created using the new one. In some cases, *domain specific transition requirements* are required. For instance, reconfiguration may be required as soon as possible yet for some live instances in particular states an exceptional treatment may be required, including repeating or roll-backing an activity.

Indeed, business process reconfiguration can be extremely challenging and can greatly benefit from automated techniques that support *i)* analysing business process requirements and transition requirements, and *ii)* constructing workflows and reconfiguration strategies satisfying these requirements.

One approach to automation is *build and verify*, in which formal verification techniques provide a sound basis for workflow analysis and can be used to ensure workflow requirement satisfaction. However, post-hoc verification requires prior construction of the workflow, and modification entails re-verification. An alternative approach is to automatically produce *correct-by-construction* workflows and reconfiguration strategies directly from requirements.

Although automatic construction of workflows from requirements has been studied (e.g., [20,11]), the *synthesis of reconfiguration strategies* for domain specific user defined transition requirements has not received attention so far.

In this paper we present a fully automated technique for business process reconfiguration based on discrete event controller synthesis. 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 a new one while satisfying any user-defined transition requirements. We discuss a translation of Dynamic Condition Response (DCR) graphs [11], a declarative language for business process requirements, into a formalism based on Labelled Transition Systems and Linear Temporal Logic [21] which is suitable for controller synthesis [7] and build upon recent work on dynamic controller update [19]. We validate the approach using three examples from the BPM Academic Initiative [1] described as DCR graphs which we reconfigured for a variety of transitions requirements.

The rest of the paper is structured as follows. Sec. 2 presents an illustrative example. Formal definitions are presented in Sec. 3. In Sec. 4 we present problems setting out how to frame it as a synthesis problem. An analysis of our technique is presented in Sec. 5. Finally, we present a discussion on related work.

2 Motivating Example

Consider a hospital process taken from a real-life study on a oncology workflow at Danish hospitals [11]. This workflow has *prescribe medicine* and *sign* activities, representing a doctor adding and signing a prescription to the patient record. In addition, a nurse, is capable of doing *give medicine* in response to the doctor prescription, or, in contrast, the nurse may indicate that they do not trust the prescription or signature by performing the *don’t trust* activity. Workflow

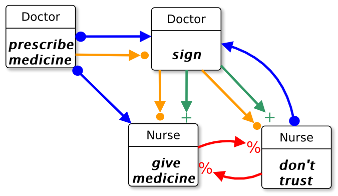

Fig. 1. DCR graph for a hospital process

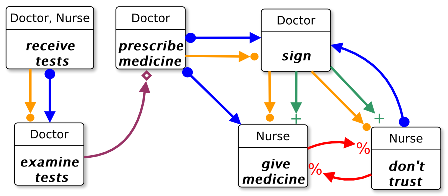

Fig. 2. DCR graph model for new hospital process.

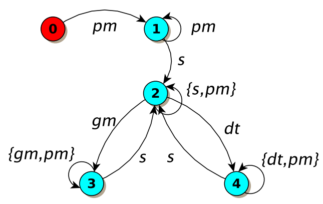

Fig. 3. Workflow for a hospital process

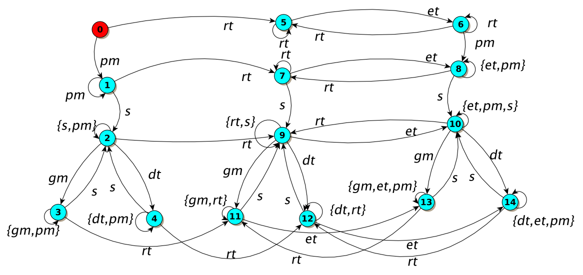

Fig. 4. Workflow for new hospital process.

requirements include that *i)* the doctor must perform *prescribe medicine* to a patient before *sign*, *ii)* the nurse can not do *give medicine* nor *don't trust* if the doctor has not done *sign*, and *iii)* the nurse can not perform both *give medicine* and *don't trust*, only one is allowed.

Fig. 1 shows these requirements modelled using Dynamic Condition Response (DCR) graphs as originally presented in [11]. A workflow that satisfies these requirements is depicted in Fig. 3 where *pm*, *s*, *gm* and *dt* labels refer to activities *prescribe medicine*, *sign*, *give medicine* and *don't trust*, respectively. The workflow is the underlying semantics of the model in Fig. 1 and can be constructed automatically using controller synthesis as described in Sec. 4.

Consider a scenario in which while patients are being treated the workflow must be changed (taken from [18]). For instance, suppose that a new internal regulation is to be put in place stating that doctors must not do *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 Fig. 2 which describes a significantly more complicated workflow (Fig. 4 where *rt* and *et* labels refer to *receive tests* and *examine tests*) that can be automatically synthesised.

A crucial decision to make is how to reconfigure a live instance running the old workflow to the new one. A naive approach would be to require an immediate [9] reconfiguration regardless of the living instance's state. Thus, if the living instance is in state 2 of Fig. 3 it should evolve to being in state 2 of Fig. 4 (i.e.,

$2 \rightsquigarrow 2$). However, this puts the patient at risk: The old workflow does not track the occurrence of *receive tests*, yet new tests may actually have been received and new prescriptions should not be done without examining them. It is safer to assume, at reconfiguration time, that tests may exist and to require be examining them (should they be available) rather than ignoring them. Consequently, it is more appropriate to update the old workflow to the new workflow according 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 allow a declarative description of transition requirements and to compute a mapping automatically. For our example, what is needed is to force *examine tests* when reconfiguring. Note that is inconsistent with both the old and new workflow requirements. In the old workflow, there is no *examine tests* activity and in the new workflow requirements *examine tests* is required after *receive tests*. Thus, what we need to express is that there is a period during the reconfiguration where neither workflow 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 modelled and how to automatically build a strategy for taking a live instance running a workflow to a new workflow guaranteeing all transition requirements.

3 Preliminaries

In this work we use Dynamic Condition Response Graphs [12] to specify business processes. To simplify presentation we use a reduced version that does not include nesting, roles, principals and roles assignments.

Definition 1. (Dynamic Condition Response Graph) *A* Dynamic Condition Response Graph *(DCR Graph) is a tuple $DG = (A, R, M)$ where A is a finite set of activities, the nodes of the graph. R is a set of graph edges. Edges are partitioned into five kinds, named and drawn as follows: conditions ($\rightarrow\bullet$), responses ($\bullet\rightarrow$), inclusions ($\rightarrow_{+}$), exclusions ($\rightarrow_{\%}$) and milestones ($\rightarrow_{\diamond}$). M is the marking of the graph. This is a triple of sets of activities (Ex, Re, In), where Ex are the previously executed, Re the currently pending and In the currently included. For all $(e, e') \in E \times E$, $e \rightarrow_{+} e'$ or $e \rightarrow_{\%} e'$ or neither of them. We denote $(\bullet\!\!\rightarrow e) = \{e' \in A \mid e' \bullet\!\!\rightarrow e\}$, $(e \bullet\!\!\rightarrow) = \{e' \in A \mid e \bullet\!\!\rightarrow e'\}$, and similarly for $\rightarrow\bullet$, $\rightarrow_{+}$, $\rightarrow_{\%}$ and $\rightarrow_{\diamond}$.*

Definition 2. (Enable activity of a DCR Graph) *Let $DG = (A, R, M)$ be a DCR graph, with $M = (Ex, Re, In)$. An activity $e \in A$ is enabled if and only if (a) $e \in In$, (b) $(In \cap (\rightarrow\!\!\bullet e)) \subseteq Ex$, and (c) $Re \cap In \cap (\rightarrow \diamond e) = \emptyset$.*

Definition 3. (Executing DCR Graph) *Let $DG = (A, R, M)$ be a DCR graph, with marking $M = (Ex, Re, In)$ and e is enabled. The result of executing e is a DCR Graph $DG' = (A, R, M')$ with $M' = (Ex', Re', In')$ such that (a) $Ex' = Ex \cup \{e\}$, (b) $Re' = (Re \setminus \{e\}) \cup (e \bullet\!\!\rightarrow)$, and (c) $In' = (In \cup (e \rightarrow_{+})) \setminus (e \rightarrow_{\%})$. We assume that initially $In = A$ and $Re = Ex = \emptyset$.*

To capture the underlying semantics of DCR graphs we use Labelled Transition Systems [14]. They are a canonical, compositional, representation of events structures ideally suited to model checking of business processes and synthesis of discrete event controllers.

Definition 4. (Labelled Transition System) *A* Labelled Transition System *(LTS) E is a tuple $(S_E,\ L_E,\ \mathit{\Delta}_E,\ e_0)$, 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, $\mathit{\Delta}_E \subseteq (S_E \times L_E \times S_E)$ is a transition relation, and $s_0 \in S_E$ is the initial state. A path of E is a sequence $\pi = s_0, \ell_0, s_1, \ell_1, s_2, \ldots$ where for every $i \geq 0$ we have $(s_i, \ell_i, s_{i+1}) \in \mathit{\Delta}_E$. A trace w is a sequence obtained by removing states from π.*

Definition 5. (Parallel Composition) *The parallel composition $E \| C$ of LTS $E = (S_E, L_E, \mathit{\Delta}_E, e_0)$ and $C = (S_C, L_C, \mathit{\Delta}_C, c_0)$ is an LTS $(S_E \times S_C, L_E \cup A_C, \mathit{\Delta}_\|, (e_0, c_0))$ such that $\mathit{\Delta}_\|$ is the smallest relation that satisfies the rules:*

$$\frac{(e,\ell,e') \in \mathit{\Delta}_E \wedge \ell \notin L_C}{((e,c),\ell,(e',c)) \in \mathit{\Delta}_\|} \quad \frac{(c,\ell,c') \in \mathit{\Delta}_C \wedge \ell \notin L_E}{((e,c),\ell,(e,c')) \in \mathit{\Delta}_\|} \quad \frac{\begin{array}{c}(e,\ell,e') \in \mathit{\Delta}_E \wedge (c,\ell,c') \in \mathit{\Delta}_C \\ \ell \in L_E \cap L_C\end{array}}{((e,c),\ell,(e',c')) \in \mathit{\Delta}_\|}$$

We use a linear temporal logic of fluents to provide a uniform framework for specifying state-based temporal properties in event-based models [10]. FLTL [10] 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 $\mathbf{X}$ (next), $\mathbf{U}$ (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 at position i, denoted $\pi, i \models f$, if and only if, one of the following conditions holds: (a) $f.Init \wedge (\forall j \in \mathbb{N} \cdot 0 \leq j \leq i \Rightarrow \ell_j \notin f.T)$, and (b) $\exists j \in \mathbb{N} \cdot (j \leq i \wedge \ell_j \in f.I) \wedge (\forall k \in \mathbb{N} \cdot j < k \leq i \Rightarrow \ell_k \notin f.T)$ 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:

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

We use $\pi \models \varphi$, instead of $\pi, 0 \models \varphi$.

Control problems aim to build an LTS that satisfies a given set of declarative requirements under certain environment conditions by having control of only a subset of the events of the environment.

Definition 6 (LTS Control [7]). *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 only blocks events in L_c, $E\|C$ is deadlock free, and $E\|C \models G$.*

Definition 7 (DCU Problem [19]). *Let $\mathcal{E} = (E, G, L_c)$ be an old specification, $\mathcal{E}' = (E', G', L_c')$ 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 a controller C_u such that: (a) $C_u \models G\ \mathbf{W}\ stopOldReq$, (b) $C_u \models T$, (c) $C_u \models \Box(startNewReq \rightarrow G')$, 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 hold G 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.

4 Dynamic Reconfiguration of Business Processes

In this section we first show how to synthesize a workflow from a DCR graph using controller synthesis Def. 6) and then show how to use dynamic controller update (Def. 7) for workflow reconfiguration.

4.1 Workflow Synthesis as a Control Problem

We now show how to extract from a DCR graph a set of controllable events L_C, an LTS E, and a FLTL formula G such that controller synthesis (Def. 6) results in a controller that enables and disables activities in such a way that its environment, as long as it only executes enabled activities, satisfies the business process requirements as described in the DCR graph. Thus L_C will contain activity enabling and disabling events, 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 workflow requirements. Finally, the formula G encodes the domain specific aspects of the DCR graph, namely the arrows that establish dependencies between activities.

Controllable and Monitorable Events The set of events that describe the control problem are defined by the activities that appear in the DCR graph (i.e., the set A). We introduce two events for each activity $a \in A$: *a.disable* and *a.happened*. The first is an event controlled by the controller. The second, is an event that will be selected by the environment (e.g., the nurse and the doctor) to indicate that the activity was executed. We say that *a.happened* is monitorable or uncontrolled. Note that we do not introduce *a.enabled*, rather we assume an

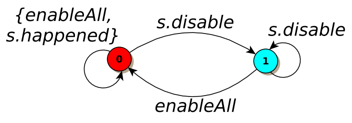

Fig. 5. *Happens*(s) LTS constraining the occurrence of *s.happened*.

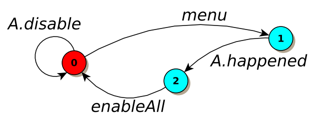

Fig. 6. *Turns* LTS constraining controller and environment turns.

event *enableAll* to reduce the number of events and states of the control problem. The controller will *enableAll* activities then select which ones to disable in such a way that if the environment executes an enabled activity, it will be consistent with the business process requirements.

We introduce one extra 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.

In conclusion, the set of controllable and uncontrollable events are $L_C = \{a.disable\,|a \in A\} \cup \{menu, enableAll\}$ and $\overline{L_C} = \{a.happened\,|a \in A\}$

Environment Model The LTS E models the two assumptions that the controller can rely upon to guarantee workflow requirements.

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 Fig. 5 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.

The second assumption is that the environment will play in turns with the controller. 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. We use only one LTS, *Turns* depicted in Fig. 6, to model this assumption. 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.

The assumptions reflect the operation of the workflow engine that will be controlled. In the hospital example, the controller first decides which activities should be enabled (*enableAll* and *a.disabled*) and then presents them to hospital staff (*menu*). It is assumed that nurses and doctor will only perform an activity if the activity is displayed by the engine, and that once performed they will report back through the engine (*a.happened*). At this point, the controller will decide again what activities to enable and update the engine display. Obviously, the *menu* event must only occur when the controller has enabled exactly all activities that if executed would not violate workflow requirements. This controller behaviour is synthesised automatically based on the formalisation of goals described next. In conclusion the LTS environment E is defined as follows $E = Turns \ \| \ Happens(a_1) \ \| \ \ldots \ \| \ Happens(a_n)$ with $A = \{a_1, \ldots, a_n\}$.

Controller Goals Goal G must model the constraints between activities that are expressed in DCR Graphs with arrows between activities. Our encoding resembles that of [20] where LTL formulas are used to formalise activity constraints of similar nature to those of DCR graphs.

We introduce three fluents for each activity $a \in A$ modelling if a belongs to sets Ex, Re, and In according to Def. 3. For simplicity, we assume that the initial marking of the DCR graph is such that $In = A$ and $Re = Ex = \emptyset$.

- $a.Executed$ models if $a \in Ex$ and is defined as $\langle\{a.happened\}, \emptyset, \bot\}\rangle$. In other words, initially no activity is in Ex and once in Ex it is never removed (see Def. 3a).
- $a.Required$ models if $a \in Re$ and is defined as $\langle\{a'.happened \mid a' \in (\bullet\!\!\rightarrow a)\}, a.happened, \bot\rangle$. That is, 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 Def. 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 Fig. 1. 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.
- $a.In$ models if $a \in In$ and is defined as $a.In = \langle\{a'.happened \mid a' \in (\rightarrow_{+} a)\}, \{a'.happened \mid a' \in (\rightarrow_{\%} a)\}, \top\rangle$, which mimics Def. 3c. Based on the relations modelled in Fig. 1, 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 Def. 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 Def. 2 we introduce for every activity $a \in A$ a formula $\alpha_a = \Box(a.happened \rightarrow a.In)$.
- For rule (b) of Def. 2 we introduce for all $a \in A$: $\beta_a = \Box(a.happened \rightarrow \bigwedge_{a' \in (\rightarrow\bullet a)} (a'.In \rightarrow a'.Executed))$. For instance, for *sign*, according to Fig. 1 we have $\beta_s = \Box(s.happened \rightarrow (pm.In \rightarrow pm.Executed))$.
- For rule (c) of Def. 2 we introduce for each $a \in A$: $\kappa_a = \Box(a.happened \rightarrow \bigwedge_{a' \in (\rightarrow_{\diamond} a)} (\neg a'.Required \vee \neg a'.In))$. For instance, $\kappa_{pm} = \Box(pm.happened \rightarrow (\neg et.Required \vee \neg et.In))$ for Fig 2.

In summary, G is defined as $\bigwedge_{a \in A} \alpha_a \wedge \beta_a \wedge \kappa_a$.

Workflow Synthesis Above we have described how to build from a DCR graph model D, the set of controllable events L_c, the LTS environment E, and the FLTL formula G that can be used to define a control problem $\mathcal{E} = (E, G, L_c)$. A solution to this problem is a controller LTS C that decides when to enable and disable activities (which correspond to events in L_c) 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: $E \| C \models G$ (Def. 6).

Note that $E \| C \models G$ 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 G. Consider a workflow for the hospital in which after *sign* only *give medicine* is enabled. The sequence *sign*, *give medicine* does not violate 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 [6] that we use for synthesis: MTSA builds eager components 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 G, thus a maximal number of activities will always be enabled.

The controllers for the DCR graphs depicted in Fig. 1 and 2 have 188 and 2291 states respectively and are too large to depict in this paper. Instead we show abstract versions of these controllers (Fig 3 and 4) in which *enableAll*, disable and *menu* events are hidden. This provides a view similar to what the Nurse and Doctor would see, only the activities that are enabled and not the controllers incremental decisions of enabling and disabling activities. Note that the abstract controllers are built automatically by MTSA tool using a hiding operator and weak bisimilarity minimisation [17].

4.2 Workflow Reconfiguration as a Dynamic Controller Update

This section is organised as follows. We first discuss how domain specific transition requirements for a workflow 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 finally how such solutions can be built automatically solving a Dynamic Controller Update problem.

Specification of Transition Requirements Recall the Hospital workflow example discussed in Sec. 2 where a transition requirement stating that activity *examine tests* should be forced when reconfiguring. More precisely, just before the moment the new business process requirements should be enforced, *examine tests* is required. The reason for requiring "just before" is that executing *examine tests* without a previous *receive tests* is not allowed in the new business process.

To formalise this transition requirement we need to refer to the moment in which the old business requirements are to be dropped (*stopOldReq*) and the moment in which the new business requirements come into force (*startNewReq*).

With these two new events the transition requirement can be formulated as follows $T_h = \Box(\mathit{stopOldReq} \rightarrow ((\bigwedge_{a \in A \setminus \{et\}} \neg a.\mathit{happened})\ \mathbf{W}\ (et.\mathit{Executed} \wedge \mathit{startNewReq})))$. Note that guaranteeing this formula requires enabling and disabling activities such that the uncontrolled events *a.happened* occur or not as required by T_h. A standard domain independent transition requirement that states that at any point one of the two business process must be adhered to (i.e., there is no transition period) can be stated as follows: $T_\emptyset = \Box((\mathit{StopOldReq} \wedge \neg \mathit{StartNewReq}) \rightarrow \bigwedge_{a \in A} \neg a.\mathit{happened})$.

Reconfiguration Workflows Returning to T_h, what would a solution to this reconfiguration problem be? Assume the workflow in Fig. 3 is in state 2, a solution to the reconfiguration is to deploy a workflow that does forces *examine tests* and then reaches a state 10 in Fig. 4. In other words, we need to build a workflow that manages the transition from the old to the new workflow, we call this workflow the *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 (Fig. 3) in state 2 to state 3. Should this happen then the reconfiguration should force *examine tests* and then move to state 13 in Fig. 4 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 Fig. 7 we depict an abstract reconfiguration workflow (enabling, disabling and *menu* events are hidden) that implements the reconfiguration from business process requirements of Fig. 1 to those of Fig. 2 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 (Fig. 3), 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 Fig. 4. The transition from the old requirements to new ones, while satisfying the transition requirements is represented by between both rectangles. Is noteworthy that there are no loops during the transition phase which guarantees that eventually the new business process requirements will be enforced.

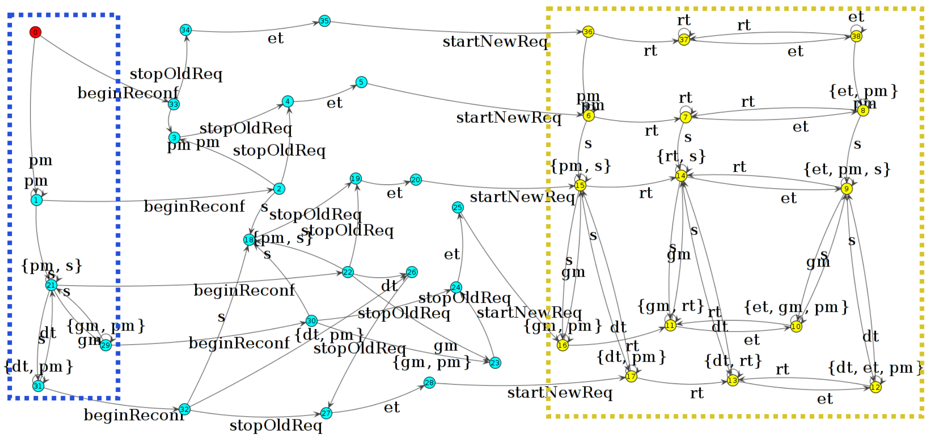

Fig. 7. Reconfiguration workflow with transition requirement T_h.

Automatic Construction of Reconfiguration Workflows Summarising, Fig. 7 represents a solution to the problem of reconfiguring business process requirements in Fig. 1 to those of Fig. 2 under transition requirement T_h. We now discuss how such solution can be built by solving a DCU problem Def. 7. The DCU problem requires two control problems $\mathcal{E} = (E, G, L_c)$ and $\mathcal{E}' = (E', G', L'_c)$ which represent in this case the old and new business process synthesis problems as described in Sec. 4.1. DCU 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 Fig.5) 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.

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 $\mathcal{E} = (E, G, L_c)$ and $\mathcal{E}' = (E', G', L'_c)$ as described in Sec. 4.1 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 Fig. 7 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 $\mathcal{E}$ and $\mathcal{E}'$ that are indi-

Case Study	# Activities	# Arrows	Transition Requirement	Reconfiguration Workflow (# States)	Minimised Reconf. Workflow (# States)
Oncology Hospital	6	13	$T_\top$	18667	54
			$T_\emptyset$	9817	34
			T_h	11155	39
			T'_h	15094	54
Doctor Assessment Process	10	25	$T_\emptyset$	22448	39
			T_D	27512	42
Insurance Process	11	25	$T_\emptyset$	15484	51
			T_I	14233	48
Computer Repair Process	18	26	$T_\emptyset$	43307	59
			T_C	52652	63

Table 1. Case Study Summary

vidually realisable yet for certain transition requirements, the update is impossible. In terms of business process reconfiguration this 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: There is no reconfiguration strategy for a live instance in which activity *prescribe medicine* has been executed.

5 Validation

The purpose of this section is to show applicability of the approach by using, in addition to motivational example, three business processes taken from BPM Academic Initiative [1] that were also modelled in the DCR Graph Tool [15]. We chose these to avoid bias in producing our own DCR graphs from workflows.

Each case study requires two DCR graphs, a source and a target for reconfiguration. We manually produced variants for each case study and used domain independent transition requirement such as $T_\emptyset$ (see Sec. 4.2) in addition to domain specific ones. All examples were run using an extension of the MTSA tool [6] and can be found at [2]. Overall, 10 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).

5.1 Oncology Hospital

This case study already discussed above is the only one for which both reconfiguration source and target DCR graphs existed. Both were taken from [18]. 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 start 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. Using a stronger domain independent transition requirement, $T_\emptyset$, the reconfiguration behaviour obtained is exactly that of an immediate reconfiguration es defined by [9].

We considered two domain specific transition requirements, T_h as discussed in Sec. 4.2 and one that delays reconfiguration when a nurse has indicated distrust regarding a patient's record: $T'_h = T_\emptyset \wedge \Box((dt.Executed \wedge \neg gm.Executed) \rightarrow \neg stopOldReq)$. As expected the resulting reconfiguration behaviour is like that of $T_\emptyset$ except that *stopOldReq* is delayed when between *don't trust* and *give medicine*.

5.2 Doctor Assessment Process

An assessment process for doctors in a hospital involves a manager asking an expert to evaluate each doctor. We used the original DCR graph as the target for reconfiguration and removed one activity to produce the source 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 $T_\emptyset$ 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, immediate reconfiguration may result in paying experts that had agreed to do a review for free in the old process. to avoid this scenario we specified the transition requirement T_D stating that if reconfiguration is requested after receiving expert review, the expert must not be paid:$T_D = T_\emptyset \wedge (\Box(startNewReq \wedge recExp.Executed) \rightarrow \Box\neg pay.Executed)$ where *recExp* is the activity representing the reception of expert review.

5.3 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 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 [27]). We solved the reconfiguration for two different transition requirements.

We used $T_\emptyset$ to compute a reconfiguration workflow which delays reconfiguration when *call the agent* has been executed but *create a new case* has not. For all other scenarios, the reconfiguration workflow do an immediate change. An alternative is to modify the target DCR graph with 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 $T_I = T_\emptyset \wedge \Box((\mathit{call.Executed} \wedge \neg \mathit{create.Executed} \wedge \mathit{startNewReq}) \rightarrow (\neg \mathit{ED}\ \mathbf{W}\ \mathit{kill.Executed}))$, where ED is the disjunction of disable events for all activities except *kill* plus *enableAll*.

5.4 Computer Repair Process

A computer repair service starts when a customer brings a defective computer. If 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*.

Initially, we solved this reconfiguration problem with the transition requirement $T_\emptyset$. As expected, executions in which the reconfiguration is requested after the budget is sent to the customer, the reconfiguration is delayed so as to not contradict the requirement of supervisor approval.

An alternative we modelled is one in which we force asking for approval for any instance in which the customer has received the budget but repair has not started. If the supervisor rejects the budget, then the customer must be contacted and apologies must be offered. The following formula (where *sup* is the activity *send to supervisor*) captures this reconfiguration requirement: $T_C = \Box((\mathit{stopOldReq} \wedge \neg \mathit{RepairStart}) \rightarrow (\neg \mathit{Happens}\ \mathbf{W}\ (\mathit{sup.Executed} \wedge \mathit{startNewReq})))$ where $\mathit{Happens}$ is the disjunction of disable events for all activities except *yes*, *no*, and *sup* plus *enableAll*.

6 Discussion and Related Work

The problem of business process reconfiguration has been studied extensively for some time [9]. [27] provides a classification of potential errors resulting from process changes. A survey of correctness criteria guaranteed by dynamic change techniques is presented in [24]. A taxonomy of reasons for reconfiguration is presented in [25] Methodological and automated support for reconfiguration has also been studied previously. Work such as [9,3,4] approach reconfiguration as a 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 [9]. A different take on reconfiguration is workflow versioning (e.g., [13,29]) where multiple workflow versions such are running simultaneously. In all cases, and in contrast to our work, the notion of a transition period in which remedial activities need to be implemented that are not compliant with the current and new workflow is not considered. The notion of reconfiguration is related to that of dynamic software updates. These have also been studied in terms of the different properties that may be expected during the update (e.g. [28]).

To reason about reconfiguration, our approach assumes a declarative specification of business process requirements (rather than an operational description in the form of a workflow). Declarative modelling approaches for business processes have been studied before. The ConDec [20] language was introduced for modelling business process based on linear temporal logic (LTL [21]). In [11], an operational semantics for a declarative graph based language is proposed and a tool [15] for enacting the underlying workflow is available. Rule based descriptions of business process requirements have also been proposed (e.g., [16,26]). Such descriptions are naturally executable. Both 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 apply a similar translation for other declarative languages.

Automatic construction of operational or executable models from declarative requirements has also been studies extensively, including work on supervisory control [23], synthesis of reactive designs [22] and automated planning [5]. This paper builds on the synthesis of discrete event controllers and in particular the work presented in [8] that uses LTS and FLTL as the input for synthesis. We strongly build on the result presented in [19] 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.

7 Conclusions

We address the problem of business process reconfiguration by providing an automatic technique that builds a reconfiguration workflow that is guaranteed to preserve any reconfiguration transition requirements provided by a user. The technique requires a declarative description of the current and new business process requirements: in this paper we start from DCR graphs, and an LTL property that describes the properties that must hold during the reconfiguration. The approach allows immediate and delayed changes, and also reconfigurations in which there is a period between business processes in which additional domain specific preparatory or remedial activities can be executed. The result is a workflow that can be hotswapped with the current one and actively manages the transition to the new business process requirements ensuring correctness.

References

1. Business process management academic initiative. `https://bpmai.org/`
2. MTSA synthesis tool and examples, `http://mtsa.dc.uba.ar`
3. van der Aalst, W.M.: Exterminating the dynamic change bug: A concrete approach to support workflow change. Information Systems Frontiers 3(3), 297–317 (2001)

4. Badouel, E., Oliver, J.: Reconfigurable nets, a class of high level Petri nets supporting dynamic changes within workflow systems. Ph.D. thesis, INRIA (1998)
5. Cimatti, A., Pistore, M., Roveri, M., Traverso, P.: Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. Artificial Intelligence 147 (2003)
6. D'Ippolito, N., Fischbein, D., Chechik, M., Uchitel, S.: Mtsa: The modal transition system analyser. In: ASE'08. pp. 475–476
7. D'Ippolito, N., Braberman, V., Piterman, N., Uchitel, S.: Synthesising non-anomalous event-based controllers for liveness goals. ACM TOSEM'13
8. D'Ippolito, N.R., Braberman, V., Piterman, N., Uchitel, S.: Synthesis of live behaviour models. In: FSE'10. pp. 77–86. ACM, New York, NY, USA
9. Ellis, C., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: COOCS'95. pp. 10–21. ACM
10. Giannakopoulou, D., Magee, J.: Fluent model checking for event-based systems. In: ESEC/SIGSOFT FSE'03. pp. 257–266. ACM, New York, NY, USA
11. Hildebrandt, T., Mukkamala, R.R.: Declarative event-based workflow as distributed dynamic condition response graphs. PLACES'10, vol. 69, pp. 59-73
12. Hildebrandt, T., Mukkamala, R.R., Slaats, T.: Nested dynamic condition response graphs. In: International conference on fundamentals of software engineering. pp. 343–350. Springer (2011)
13. Kradolfer, M., Geppert, A.: Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In: Int. J. Coop. Info. Syst. 1999
14. Magee, J., Kramer, J.: State models and java programs. wiley Hoboken (1999)
15. Marquard, M., Shahzad, M., Slaats, T.: Web-based modelling and collaborative simulation of declarative processes. In: International Conference on Business Process Management. pp. 209–225. Springer (2016)
16. Mejia Bernal, J.F., Falcarin, P., Morisio, M., Dai, J.: Dynamic context-aware business process: a rule-based approach supported by pattern identification. In: SAC'10. pp. 470–474
17. Milner, R.: A calculus of communicating systems. LNCS 92 (1980)
18. Mukkamala, R.R.: A Formal Model For Declarative Workflows. Ph.D. thesis, IT University of Copenhagen (2012)
19. Nahabedian, L., Braberman, V., D'Ippolito, N., Honiden, S., Kramer, J., Tei, K., Uchitel, S.: Dynamic update of discrete event controllers. IEEE TSE'18 pp. 1–1
20. Pesic, M., Van der Aalst, W.M.: A declarative approach for flexible business processes management. In: BPM'06. pp. 169–180
21. Pnueli, A.: The temporal logic of programs. In: FOCS'77. pp. 46–57
22. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL'89
23. Ramadge, P.J., Wonham, W.M.: The control of discrete event systems. Proc. of the IEEE 77(1), 81–98 (1989)
24. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems—-a survey. Data & Knowledge Engineering 50(1), 9–34 (2004)
25. Schonenberg, H., Mans, R., Russell, N., Mulyar, N., van der Aalst, W.M.: Towards a taxonomy of process flexibility. In: CAiSE'08. vol. 344, pp. 81–84
26. Vasilecas, O., Kalibatiene, D., Lavbič, D.: Rule-and context-based dynamic business process modelling and simulation. Journal of Systems and Software 2016
27. V.D Aalst, W.M., Stefan, J.: Dealing with workflow change: identification of issues and solutions. CSSE'00 15(5), 267–276
28. Zhang, J., Cheng, B.H.: Model-based development of dynamically adaptive software. In: ICSE'06. pp. 371–380
29. Zhao, X., Liu, C.: Version management in the business process change context. In: BPM'07. pp. 198–213