Dynamic Update of Discrete Event Controllers

L. Nahabedian⋆, V. Braberman⋆, N. D'Ippolito⋆, S. Honiden+, J. Kramer†, K. Tei⋆ and S. Uchitel†⋆

Abstract—Discrete event controllers are at the heart of many software systems that require continuous operation. Changing these controllers at runtime to cope with changes in its execution environment or system requirements change is a challenging open problem. In this paper we address the problem of dynamic update of controllers in reactive systems. We present a general approach to specifying correctness criteria for dynamic update and a technique for automatically computing a controller that handles the transition from the old to the new specification, assuring that the system will reach a state in which such a transition can correctly occur and in which the underlying system architecture can reconfigure. Our solution uses discrete event controller synthesis to automatically build a controller that guarantees both progress towards update and safe update.

Index Terms—Controller Synthesis, Dynamic Update, Adaptive Systems.

1 Introduction

Dynamic software update is a key feature for systems that require continuous operation. Such a requirement is commonplace in many domains including safety, mission and business critical systems. Continuous operation entails runtime update of systems to account for changes in the execution environment and in the requirements they are expected to achieve. Engineering systems such that they be changed in a sound, predictable manner without stopping or disrupting their operation is technically challenging and has been addressed in complementary ways, including dynamic component update and reconfiguration (e.g. [1]), controller reconfiguration (e.g., [2, 3, 4]) and more recently in the design of adaptive systems [5].

Consider, for instance, workflow management software systems aimed to support continuous operation. Such systems have an explicit runtime representation of the procedure by which the system supports achieving the organisation's goals (i.e, the workflow). Deployed in organisations whose business evolves beyond the scenarios known at the time of design, development or deployment time, these systems are built with the premise that they are long-running and that they must support new specifications [6]. They need to provide support for runtime modification of the workflow to accomodate changes in business goals and interfaces to other systems, increased regulatory procedures, weakened assumptions on potential fraudulent activity, and so on. Similarly, many reactive systems have a controller component that enacts a strategy (abstractly represented as a state machine) that coordinates multiple components and services to achieve mission goals [7]. Changes at runtime in the execution environment and/or in the requirements need runtime mechanisms to compute a new controller and hot-swap it so as to guaranteeing the correct behaviour of the system throughout the transition from the old system specification to the new one.

In this paper we formulate and address the problem of *dynamic controller update (DCU)* which describes how a system must cope with specification changes in the requirements and/or the environment. This problem can be cast as automatically building an *update controller* that manages the hot swap of the current controller (that guarantees the old specification) with a new one (that is to guarantee the new specification). The update controller must first take control of the system. Then, whilst satisfying the old specification, it must guide the system (ensuring progress) towards a state in which both reconfiguration and the change from guaranteeing the old specification to the new one can occur, ensuring that transitional requirements (if any) are satisfied. Finally it must operate guaranteeing the new specification.

Our approach to *solving the DCU problem* is based on discrete event controller synthesis (e.g. [8, 9, 10]). Controller synthesis automatically builds an operational strategy (which can be represented compactly as a state machine) that monitors events and command actions in such a way that it guarantees a given goal under given environment assumptions. Although controller synthesis for general linear temporal logic is EXP2TIME complete, the restriction of specifications to safety properties results in goals for the controller synthesis to be limited to safety and reachability properties, thus retaining linear time complexity.

The main contributions of this paper are twofold:

- a precise formulation of the dynamic controller update problem that includes architectural reconfiguration and supports flexible transition requirements, and
- a solution to the update problem based on controller synthesis that not only guarantees progress towards update, but guarantees satisfaction of the transition requirements.

In particular, we *i)* show how transition requirements between specification changes can be described, restricting when the change can occur and when any reconfiguration needed should occur; *ii)* formalise the dynamic controller update problem, showing correctness, soundness and completeness; and *iii)* define an automatic procedure, based on discrete event control theory, for computing an update controller that handles hot swapping, reconfiguration and the new specification. The input of the synthesis procedure

⋆ *Universidad de Buenos Aires, FCEyN, ICC, CONICET, Argentina.*
+ *National Institute of Informatics, Japan.*
† *Department of Computing, Imperial College, London, UK.*
Manuscript received DATE; revised DATE.
This work was partially supported by ANPCYT, UBACYT, and CONICET.

is the current controller and its specification, the transition requirements, the new specification to be satisfied and a state mapping between the current and the new specification. The output of the procedure is a controller that can be hotswapped with the current controller at any point in time, that continues to satisfy the old specification but guides the system to a safe transition state and then guarantees the new specification. Alternatively, if it is not possible to control the running system to move from one specification to another without violating the transition requirements or one of the specifications, then no controller is synthesised. The synthesis procedure is complete, hence if there is no strategy that can guide the current system to a safe state to allow a safe update with respect to transition requirements and the new specification, then, the procedure reports so.

The problem of dynamic update has been studied extensively. The problem defined in this paper builds on, and extends, existing work in a number of ways. We build on the need to have explicit, user specified, requirements for controller transitions as in [11, 12] but compute an update strategy for these requirements automatically. Automatic strategy computation has been addressed before (e.g., [13, 14, 15]) for more restricted transition requirements. Compared to these approaches a noteworthy distinction is that we drop the assumption that the system to be updated will reach, on its own, a state in which it is safe to perform the update. We use discrete event controller synthesis to automatically build a controller that guarantees both progress towards update and safe update.

This paper extends our previous work [16] in three major ways. Firstly, we overcome a limitation of that formalisation that leads to increasingly complex specifications. In [16] the new specification to be enforced is required to include all the propositions of the old specification. This means that as the specification evolves in time, old propositions cannot be discarded, leading to bloated specifications and incurring in additional computational cost. In this paper, to avoid bloating, we recast the formalisation of the controller update problem based on Labelled Transition System and Fluent Linear Temporal Logic instead of using Labelled transition Kripke Structures (see Section 3). We also allow a more general mechanism for specifying how states from the old specification are mapped to states in the new one. Second, we provide a proof showing soundness and completeness of the approach. Finally, we provide more detailed and comprehensive validation and evaluation that includes amongst others an industrial case study from the workflow management domain.

The rest of the paper is structured as follows. Section 2 presents an illustrative example based on a production cell. Then, formal definitions are presented in Section 3. These are required to formalize the problem of dynamic controller update in Section 4. Later, in Section 5, we propose a solution to this problem giving a proof of correctness and completeness. Validation is presented in Section 6. Finally, we present a discussion and related work and then conclude.

2 Illustrative Example: Production Cell

Consider an industrial automation scenario [17] in which a robotic arm applies various tools to raw products taken from an In tray and then drops the finished products on the Out tray. The operation of the factory is driven by a software controller that sequences commands adhering to a specification E, G and A. A is the set of events the controller can execute, for instance *drill*, *polish*, *clean*, and *stamp*. E models the assumptions that the controller can rely on to achieve its goals. E may include, for instance, that the paint tool once commanded to polish product x ($polish(x)$) will respond with $polishOK(x)$ or $polishNOK(x)$ representing success or failure in polishing x. Such an assumption can be easily modelled with an automaton. Finally, G models the goals for the controller. For instance, G may require a product to be placed in the Out tray only if it has been cleaned, polished and drilled (in that order) and no errors have occurred, or alternatively if an error has occurred and it must have been stamped as faulty. A formalisation using the linear temporal logic of fluents (see FLTL in section 3) of some of the goals may be:

$$\begin{aligned}
&\mathit{ToolOrder} \equiv \Box\, \forall x \cdot (\mathit{Cleaned(x)} \Rightarrow \mathit{Polished(x)}) \wedge \\
&\qquad\qquad \mathit{Polished(x)} \Rightarrow \mathit{Drilled(x)}) \\
&\mathit{ToolsRequired} \equiv \Box\, \forall x \cdot out(x) \Rightarrow (\mathit{Faulty(x)} \vee \\
&\quad (\mathit{Drilled(x)} \wedge \mathit{Polished(x)} \wedge \mathit{Cleaned(x)} \wedge \neg \mathit{Stamped(x)})) \\
&\mathit{NoProcessingIfFaulty} \equiv \Box\, \forall x \cdot (\mathit{Faulty(x)} \Rightarrow \\
&\qquad\qquad \neg(\mathit{drill(x)} \vee \mathit{polish(x)} \vee \mathit{clean(x)}))
\end{aligned}$$

Consider a scenario in which while the factory is processing products it is decided that the production process must be changed. This decision may be taken due to many different factors: the set of available tools changes (e.g., a tool breaks, or a new tool is introduced), the specification of how to process a product type changes (e.g., new business rules), or other constraints change (e.g., a new energy consumption requirement constrains the concurrent use of certain tools). A simple solution to this problem is to wait for the production line to be empty (i.e., wait for all products to be processed and moved to the out tray), stop the plant, change the controller and then restart the plant. An off-line update such as this one may be unacceptable where factory down-time has serious economic consequences.

Assume that for business reasons, a polishing tool is to be exchanged for a paint tool where the change entails a re-ordering in the production workflow. The new production workflow is captured by specification E', G' and A' where A' no longer has *polish* but has *paint* instead. For instance, E' will include assumptions on how the Paint tool works and G' may have the revised goals:

$$\begin{aligned}
&\mathit{ToolOrder'} \equiv \Box\, \forall x \cdot (\mathit{Drilled(x)} \Rightarrow \boldsymbol{Painted(x)}) \wedge \\
&\qquad\qquad (\boldsymbol{Painted(x)} \Rightarrow \mathit{Cleaned(x)}) \\
&\mathit{ToolsRequired'} \equiv \Box\, \forall x \cdot out(x) \Rightarrow (\mathit{Faulty(x)} \vee \\
&\quad (\mathit{Drilled(x)} \wedge \boldsymbol{Painted(x)} \wedge \mathit{Cleaned(x)} \wedge \neg \mathit{Stamped(x)}))
\end{aligned}$$

How should the current controller be updated to satisfy the new specification? When is it safe to swap controllers? What strategy should the new controller use once it is in place? When can the Paint tool driver be instantiated and bound into the current software architecture? When can the Polish tool be removed from the current architectural configuration? The answers to these questions are domain specific.

As explained in [18], the answers to these questions are transition requirements that must be provided by domain experts.

For instance, what should be done with products that have been partially processed according to G? Perhaps they should be finished off according to the new requirements expressed in G'? Should a polished but not clean product be cleaned and then placed on the Out tray? Should it be discarded without further processing? Or should the update be delayed until there are no polished products on the line?

To specify transition requirements, we must first define what a transition is and how to refer to it. For this, assume that within the update process there will be a command to signal when the old specification is dropped (*stopOldSpec*) and an event to signal from when the new specification is to be guaranteed (*startNewSpec*). For indicating if these events have occurred, we use fluents *OldSpecStopped* and *NewSpecStarted*.

One possible transition requirement is that no polished product should be on the line:

$T_1 = \textit{startNewSpec} \Rightarrow \neg\textit{OldSpecStopped} \wedge (\forall x \cdot \textit{OnProductionLine(x)} \Rightarrow \neg\textit{Polished(x)})$

Another transition requirement could be that products are to be either output according to the new specification G' or stamped for trashing. This allows, for instance, partially processed products that cannot be continued to be processed according to G'.

$$\begin{aligned}T_2 \equiv\ & (\textit{OldSpecStopped} \wedge \neg\textit{NewSpecStarted}) \Rightarrow (\textit{ToolOrder}' \wedge \\ & \quad \textit{NoProcessingIfFaulty} \wedge \textit{ToolsRequired}'' \wedge \ldots) \\ \textit{ToolsRequired}'' \equiv\ & \Box\forall x \cdot out(x) \Rightarrow (\textit{Faulty(x)} \vee \textit{Stamped(x)} \vee \\ & \quad (\textit{Drilled(x)} \wedge \textit{Painted(x)} \wedge \textit{Cleaned(x)} \wedge \neg\textit{Polished(x)}))\end{aligned}$$

Returning to the problem of when to reconfigure, T_1 may also include a requirement disallowing reconfiguration when the Polish tool is working on a product (i.e., $reconfigure \implies \forall x \cdot \neg BeingPolished(x)$). Such a requirement would ensure that the command reconfigure (which will bind the paint tool driver and unbind the polish tool driver) is issued safely.

Note that if T_1 were selected, an interesting liveness problem may arise. It may be the case that there is always a polished product on the line: if new products arrive regularly and the current controller sends them to be polished before existing polished products on the line are drilled and placed in the Out tray, the reconfigure command can never be issued as it would violate T_1. The current controller needs to be guided to a state in which the update can occur. In fact, it must be stopped from further polishing and forced to finish off any already polished products.

The fact that the current controller needs to be guided to an updatable state shows that a controller update strategy requires replacing the current controller with another one that can continue to satisfy G (e.g., finishing off polished products according to G) while ensuring that eventually an update state is reached (e.g., no polished products on the line). Thus, the solution to how the system is updated from E, G and A; to E', G' and A' also satisfying T_1 is to have an update controller that replaces the current controller, guides the system to states in which it can reconfigure, can signal that the old specification is dropped and the new one has started without violating T_1.

Indeed, we present a fully automated technique that can guarantee a correct update of the controller for the production plant. Informally, the input to the technique we present is the current specification E, G and A, the controller currently supervising the production plant (C), the new specification E', G' and A', and transition requirements (T). The output (should there be a solution to the problem) is an update controller (C') that assures that the resulting system satisfies the following requirements:

(i) C can be hot-swapped by C' at any point in time.
(ii) G will continue to hold until C' signals *stopOldSpec*.
(iii) T that prescribes *stopOldSpec*, *startNewSpec* and *reconfigure* holds.
(iv) G' will hold once C' signals *startNewSpec*.
(v) Once C and C' are hot-swapped, the following will eventually occur: *reconfigure*, *startNewSpec* and *stopOldSpec*.

A schematic diagram showing examples of dynamic controller update for three different transition requirements is given in Figure 1. We depict main update events (*hotSwap*, *reconfigure*, *stopOldSpec* and *startNewSpec*), how *hotSwap* and *reconfigure* change the running system (from $C\|E$ to $C'\|E$ and from $C'\|E$ to $C'\|E'$ respectively), and when goals G and G' hold.

Note that the computation of the update controller can be performed while the system is in operation. An update scenario would proceed as follows: The plant is being controlled by C to satisfy G when a decision is made to change the production process. This may occur, for example, because some quality check on finished products fails and a problem can be traced back to the polisher, or some other business concerns arise. Such a decision may be the result of human intervention or may be part of, for example, the Monitor and Analysis phases of a MAPE loop [19] in an adaptive system.

Next, a decision on what to do in the face of this unexpected problem must be made. In our scenario a decision would result in G' and T. Again, this decision may (and in a production plant is likely to) be done manually, but could also be the result of an automatic or semi-automatic plan phase of an adaptive systems' MAPE loop. Our technique computes an update controller C' that would be hot-swapped in, removing C and setting the initial state of C' according to the current state of C.

Controller C' executes a strategy that satisfies the transition requirement T for any possible state of the plant (e.g., number of partially processed products and the particular stage of the production process each one is in). For instance, it will trash partially processed products that cannot be further processed to satisfy G'; *reconfigure* the system binding the paint tool into the production line; continue processing partially processed products that can be further processed to satisfy G'; and process all new products that come through the In tray according to G'.

3 Formal Foundations

In this section we present the background to allow formalisation of the problem of dynamic controller update and for presenting the proposed rigorous solution based on controller synthesis.

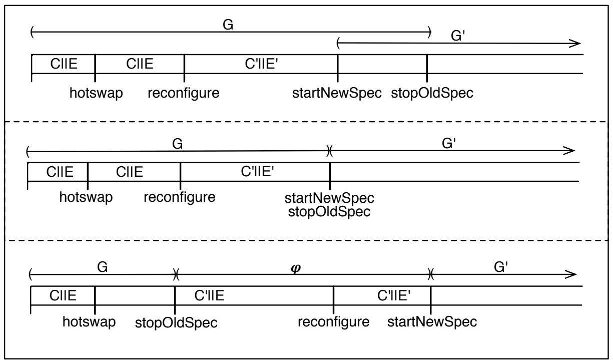


Fig. 1: Dynamic controller updates with different transition requirements (from top to bottom): *stopOldSpec* $\Rightarrow$ *NewSpecStarted*, *startNewSpec* $\Rightarrow$ *OldSpecStopped* and $(\mathit{OldSpecStopped} \wedge \neg \mathit{NewSpecStarted}) \Rightarrow \varphi$, where φ is a safety property.

3.1 Labelled Transition Systems

Labelled Transition Systems are a canonical, compositional, representation of reactive systems.

Definition 3.1. *(*Labelled Transition System*) A* Labelled Transition System *(LTS) E is a tuple $(S_E,\ A_E,\ \Delta_E,\ e_0)$, where S_E is a finite set of states, $A_E \subseteq$ Act is its* communicating alphabet*, Act is the universe of all observable events, $\Delta_E \subseteq (S_E \times A_E \times S_E)$ is a transition relation, and $s_0 \in S_E$ is the initial state. We say that E is deterministic if $(e, \ell, e') \in \Delta_E$ and $(e, \ell, e'') \in \Delta_E$, then, $e' = e''$, and is deadlock-free if for all $e \in S$ there exists $(e, \ell, e') \in \Delta_E$.*

Notation 1. *Let E be an LTS, for a state $e \in S_E$, we denote $\Delta_E(e) = \{\ell \mid (e, \ell, e') \in \Delta_E\}$.*

Notation 2. *Let E be an LTS, for a state $e \in S_E$, we denote changing the initial state of E from s_0 to e as $E(e)$.*

Definition 3.2. *(*Trace*) A trace of an LTS E is a sequence of labels $\pi = \ell_0, \ell_1, \ldots$, for which there exists a sequence of states $s_0, s_1, \ldots$ such that s_0 is the initial state of E and $\forall i \geq 0 \cdot \ell_i \in \Delta_E(s_i)$. We denote by $\pi \in E$ a trace on E and by $\ell \in \pi$ a label in a trace π.*

We introduce the following equivalence relations between LTS.

Definition 3.3. *(*Isomorphism*) Let $E = (S_E,\ A,\ \Delta_E,\ s_E)$ and $M = (S_M,\ A,\ \Delta_M,\ s_M)$ be LTS. An isomophism is a bijection $f : S_E \to S_M$ that preserves transitions:*

$$(q, \ell, q') \in \Delta_E \quad \textit{if and only if} \quad (f(q), \ell, f(q')) \in \Delta_M$$

for all $q, q' \in S_E$. If there exists an isomorphism between E and M, then we say that E and M are isomorphic, denoted $E = M$.

Definition 3.4. *(*Bisimulation*) Let $\mathcal{P}$ be the universe of all LTS. A binary relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a bisimulation if and only if whenever $(P, Q) \in \mathcal{R}$ then for each $a \in Act$ the following hold:*

- *if $(P \xrightarrow{a} P')$, then, $(\exists Q' \cdot Q \xrightarrow{a} Q' \wedge (P', Q') \in \mathcal{R})$*
- *if $(Q \xrightarrow{a} Q')$, then, $(\exists P' \cdot P \xrightarrow{a} P' \wedge (P', Q') \in \mathcal{R})$*

where $X \xrightarrow{x} X'$ denotes that the LTS X can execute x reaching a state x' and $X' = X(x')$. We denote by $P \sim Q$ that P and Q are bisimilar.

Reactive systems are built compositionally. Such composition is often modeled with LTS as the cartesian product of component LTS states where communication is modeled as synchronous communication on shared events and propositions. We formalise the parallel composition as follows.

Definition 3.5. *(*Parallel Composition*) The parallel composition $E \| C$ of two LTS $E = (S_E, A_E, \Delta_E, e_0)$ and $C = (S_C, A_C, \Delta_C, c_0)$ is an LTS $(S_E \times S_C, A_E \cup A_C, \Delta_{\|}, (e_0, c_0))$ such that $\Delta_{\|}$ is the smallest relation that satisfies the rules below:*

$$\frac{(e, \ell, e') \in \Delta_E}{((e, c), \ell, (e', c)) \in \Delta_{\|}}\ \ell \notin A_C$$

$$\frac{(c, \ell, c') \in \Delta_C}{((e, c), \ell, (e, c')) \in \Delta_{\|}}\ \ell \notin A_E$$

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

The relabelling operation defines an LTS by relabelling or removing transitions from another LTS. It does so by either removing a transition when its label is not defined in the relabelling function, or by changing the label according to the relabelling function.

Definition 3.6. (Relabelling operator) *Let $E = (S_E, A_E, \Delta_E, e_0)$ be an LTS and $f : A_E \to$ Act be an injective partial function. The relabelling operation $[E]_f$ is an LTS (S_E, A', Δ_f, e_0), where Δ_f is the smallest relation that satisfies the rule below:*

$$\frac{(e, \ell, e')\ \in\ \Delta_E}{(e, f(\ell), e')\ \in\ \Delta_f}$$

Property 3.1. *Let E, B be deterministic LTS, and $f : Act \to Act$ be a partial injective function, then, the following equivalence holds: $[E \| B]_f \sim [E]_f \| [B]_f$*

We introduce an interrupt handler operation that defines the sequential execution of two LTS where the switch from one LTS to the other is triggered by an event (α).

Definition 3.7. (Interrupt handler) *Let $E = (S_E, A_E, \Delta_E, e_0)$ and $N = (S_N, A_N, \Delta_N, n_0)$ be LTS, H be an interrupt handler relation such that $H \subseteq (S_E \times S_N)$, and α be an interrupt event such that $\alpha \notin (A_E \cup A_N)$,.*

The interrupt handler $E \lightning^{\alpha}_{H} N$ is an LTS defined as $(S_E \cup S_N, A_E \cup A_N \cup \{\alpha\}, \Delta_{\lightning}, e_0)$, where $\Delta_{\lightning}$ is the smallest relation that satisfies the rules below:

$$\frac{(e, \ell, e')\ \in\ \Delta_E}{(e, \ell, e')\ \in\ \Delta_{\lightning}} \quad \frac{(n, \ell, n')\ \in\ \Delta_N}{(n, \ell, n')\ \in\ \Delta_{\lightning}}$$

$$\frac{}{(e, \alpha, n)\ \in\ \Delta_{\lightning}}\ (e, n)\ \in\ H$$

Property 3.2. *Let A, B, C and D be LTS, H and K be interrupt handlers relations and ℓ be an iterrupt event, then, the following equivalence holds:*

$$(A \lightning^{\alpha}_{H} B) \mid\mid (C \lightning^{\alpha}_{K} D) \sim ((A||C) \lightning^{\alpha}_{H'} B) \mid\mid ((A||C) \lightning^{\alpha}_{K'} D)$$

where $((a, c), b) \in H'$ iff $(a, b) \in H$, and, $((a, c), d) \in K'$ iff $(c, d) \in K$.

3.2 Fluent Linear Temporal Logic

Linear Temporal Logic (LTL) is often used to describe behaviour requirements [20, 21]. The motivation for choosing

$$\begin{array}{lcl}
\pi, i \models_d \neg\varphi & \triangleq & \pi, i \not\models_d \varphi \\
\pi, i \models_d \varphi \vee \psi & \triangleq & (\pi, i \models_d \varphi) \vee (\pi, i \models_d \psi) \\
\pi, i \models_d \mathbf{X}\varphi & \triangleq & \pi, i+1 \models_d \varphi \\
\pi, i \models_d \varphi \mathbf{U} \psi & \triangleq & \exists j \geq i \cdot \pi, j \models_d \psi \quad \wedge \\
 & & \quad \forall\, i \leq k < j \cdot \pi, k \models_d \varphi
\end{array}$$

Fig. 2: Semantics for the satisfaction operator.

an LTL of fluents is that it provides a uniform framework for specifying state-based temporal properties in event-based models [21]. FLTL [21] 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 actions, $f.T$ is a set of terminating actions and $f.I \cap f.T = \emptyset$. A fluent may be initially *true* or *false* as indicated by the $f.Init$. Note that, every action $\ell \in Act$ induces a fluent, namely $\dot{\ell} = \langle\ \ell,\ Act \setminus \{\ell\},\ \bot \rangle$. Finally, the alphabet of a fluent is the union of its terminating and initiating actions.

Let $\mathcal{F}$ be the set of all possible fluents and d be a fluent definition function $d : \mathcal{F} \to \langle I, T, Init \rangle$. An FLTL formula is defined inductively using the standard Boolean conectives 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 $\varphi\mathbf{U}\psi \vee \Box\varphi$.

Let Π be the set of infinite traces over *Act*. 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:

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

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

Given an infinite trace π, the satisfaction of a formula φ at position i for a fluent definition d, denoted $\pi, i \models_d \varphi$, is defined as shown in Figure 2. We say that φ holds in π for a fluent definition d, denoted $\pi \models_d \varphi$, if $\pi, 0 \models_d \varphi$.

To support combined reasoning over two specifications (the old and new) that may have different scopes (i.e. refer to different sets of events) we introduce a fluent definition extension and a related property.

Definition 3.8. (Fluent definitions extension) *Let d and d' be fluent definitions over Σ and Σ' respectively, d_e is a fluent definition extension of d and d', if and only if, the following conditions hold:*

- ► $\forall\, (f_e, \langle I_e, T_e, Init_e \rangle) \in d_e, \exists\, (f, \langle I, T, Init \rangle) \in d$ *such that* $I_e = I \cup \dot{I}$ *and* $T_e = T \cup \dot{T}$ *where* $\dot{I} \subseteq (\Sigma' \setminus \Sigma)$ *and* $\dot{T} \subseteq (\Sigma' \setminus \Sigma)$.
- ► $\forall\, (f, \langle I, T, Init \rangle) \in d, \exists\, (f_e, \langle I_e, T_e, Init_e \rangle) \in d_e$ *such that* $I_e = I \cup \dot{I}$ *and* $T_e = T \cup \dot{T}$ *where* $\dot{I} \subseteq (\Sigma' \setminus \Sigma)$ *and* $\dot{T} \subseteq (\Sigma' \setminus \Sigma)$.
- ► $\forall\, (f_e, \langle I_e, T_e, Init_e \rangle) \in d_e, \exists\, (f', \langle I', T', Init' \rangle) \in d'$ *such that* $I_e = I' \cup \dot{I}$ *and* $T_e = T' \cup \dot{T}$ *where* $\dot{I} \subseteq (\Sigma \setminus \Sigma')$ *and* $\dot{T} \subseteq (\Sigma \setminus \Sigma')$.
- ► $\forall\, (f', \langle I', T', Init' \rangle) \in d', \exists\, (f_e, \langle I_e, T_e, Init_e \rangle) \in d_e$ *such that* $I_e = I' \cup \dot{I}$ *and* $T_e = T' \cup \dot{T}$ *where* $\dot{I} \subseteq (\Sigma \setminus \Sigma')$ *and* $\dot{T} \subseteq (\Sigma \setminus \Sigma')$.

3.3 Labelled Transition Systems Control Problem

The notion of legality (based on Interface Automata [22]) 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 3.9. (Legal LTS) *Let $P = (S_P, A_P, \Delta_P, p_0)$ and $Q = (S_Q, A_Q, \Delta_Q, q_0)$ be LTS, $C \subseteq (A_P \cup A_Q)$ be a set of events that P does control and $U \subseteq (A_P \cup A_Q)$ 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 [22]. Here, we request the P not to be more robust (i.e. accept more uncontrollable events) than Q can exhibit.

An LTS control problem can be described as follows: Given an LTS that describes the behaviour of the environment, a set of controllable actions, an FLTL formula as the goal for the controller, and a fluent definition, the LTS control problem is to find an LTS that only restricts the occurrence of controllable actions and guarantees that the parallel composition between the environment and the LTS controller is deadlock free and satisfies the goal.

Definition 3.10 (LTS Control [10]). *Let $E = (S_E, A_E, \Delta_E, e_0)$ be an environment model in the form of an LTS, $A \subseteq A_E$ be a set of controllable actions, $\widehat{A} \subseteq A_E$ be a set of uncontrollable actions, G be a controller goal in the form of an FLTL property, and d a fluent definition. A solution for the LTS control problem with specification $\mathcal{E} = (E, G, d, A, \widehat{A})$ is an LTS $C = (S_C, A_E, \Delta_C, c_0)$ such that C is a legal LTS for E with respect to $(A, \widehat{A})$, $E\|C$ is deadlock free, and $E\|C \models_d G$.*

4 Problem Statement

In this section we precisely state the problem of dynamic update of discrete event controllers. In other words, we formalise what it means for an update controller to be hot swapped into a system at runtime while guaranteeing that the resulting system will eventually transition correctly to the new system specification. We first formalise controller hot-swapping, then environment reconfiguration and finally a notion of correctness.

4.1 Hot-swapping Controllers

We now formally define the behaviour of a system in which the current controller is hot swapped by another one. Assume a controller C is enacted within an environment E, i.e. a system $C\|E$. Suppose that the controller C is to be hot swapped with a new controller C'. In some cases,

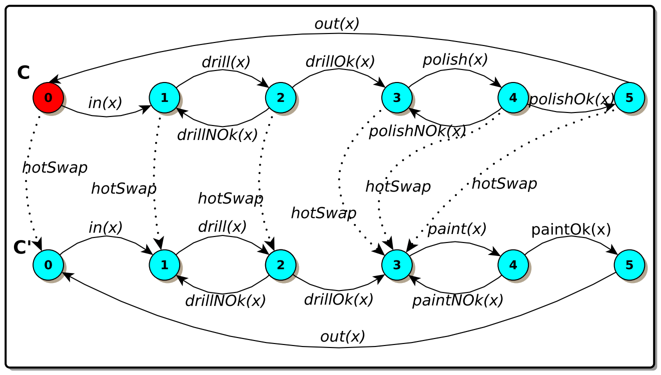


Fig. 3: Hotswap from (a reduced – two tool) production cell controller to another (i.e., neither have the cleaner tool). ($C \lightning_H^{hotSwap} C'$). Dotted lines represent hotswap transitions induced by H.

the initial state of C' at the time in which it is swapped in can be fixed at design time. However, in general the initial state of C' depends on the state of the system at the instant at which C' takes control. Thus, we support the modelling of a controller hot swap with a mapping from the states of C to C' that is user provided and sets the initial state of the controller C'. We model the hot-swap of C with C' as $(C \lightning_H^{hotSwap} C') \| E$ using the interrupt handler.

We assume that *hotSwap* is not in the alphabet of C, C' and E, and that H covers all states in C. Consequently, in $(C \lightning_H^{hotSwap} C') \| E$ the event *hotSwap* is unconstrained and can occur only once but at any point. Indeed, this models that *hotSwap* will be triggered by the underlying enactment infrastructure which will use H to set the current state of C' according to that of C.

At the top of Figure 3 we depict a controller C for a reduced (i.e, two tool) version of the production cell with slightly modified objectives (reduced to aid understandability of the diagram): after receiving a raw element x in the tray (*in(x)*), the controller will *drill(x)* and *polish(x)* the element. The *out(x)* command will be executed by the controller when both tools were applied to the element correctly. At the bottom of the same figure, we show a controller C' that instead of polishing elements, it will paint them (*paint(x)*) instead. The combination of both controllers plus the dotted *hotSwap* transitions is the result of $(C \lightning_H^{hotSwap} C')$, where the *hotSwap* transitions coincide with relation H from states in C to states in C' (i.e $H = \{(0,0),(1,1),(2,2),(3,3),(4,3),(5,3)\}$).

It is important to note that the communicating alphabet of $C \lightning_H^{hotSwap} C'$ is a superset of the alphabets of C and C'. Thus, an event ℓ that is in the alphabet of C that is not in that of C' will be restricted from occurring after *hotSwap* in the composition $(C \lightning_H^{hotSwap} C')$ because there is no transition in C' labelled with ℓ (see parallel composition, Definition 3.5). The same holds for events in the alphabet of C' that are not in the alphabet of C, where such events will be constrained from occurring in $(C \lightning_H^{hotSwap} C')$ before *hotSwap*. For instance, in the example, C has in its alphabet *polish(x)* but not *paint(x)* which is in the alphabet of C'. The term $C \lightning_H^{hotSwap} C'$ will prohibit the occurrence of *paint(x)* before *hotSwap* and the occurrence of *polish(x)* after hotSwap.

4.2 Controlling Reconfiguration

Controllers are updated in systems because the system goals or the environment assumptions have changed. The latter may be due to adoption of unrealistic or invalid assumptions or because the executing (software and/or hardware) configuration environment of the controller has changed: an API that provides services to the controller needs to be changed, a sensor or actuator is not working with the desired precision. In the case of our example, the required change in the configuration environment is that the Polishing tool is no longer needed and Paint tool is. This involves unloading the driver for the Polish tool and loading the driver for the Paint tool.

A change in the executing configuration environment typically needs to be managed and coordinated with the overall behaviour that is satisfying the system goals. Thus, the controller that manages the dynamic update needs to be able to control when reconfiguration should occur, and it must signal the change at a point in time where the whole update can be guaranteed to be correct. Note that the latter setting guarantees planned environment changes where the update controller chooses when to reconfigure.

We model a controller that is capable of changing its configuration dynamically with the term $C' \| (E \lightning_R^{reconfigure} E')$ assuming that *reconfigure* is part of the communicating alphabet of C'. This means that, as opposed to *hotSwap*, *reconfigure* is controlled by the controller C'. Note that in this way we are encapsulating a possibly complex reconfiguration procedure, with multiple steps, as an atomic reconfigure action. Should an atomic reconfiguration not be a reasonable assumption, then it is possible to model, in the spirit of [23, 24], within E' the reconfiguration steps and include it as part of the control problem.

The relation R, sets the initial state of E' according to the current state of E at the time of *reconfigure*. One possible choice for R is to map all states of E to one fixed initial state of E' modelling fixed initialisation of the environment. Note that we do not require R to be defined for all states of E; this provides a way of restricting when C' is allowed to reconfigure.

It is important to note that R may introduce non-determinism by mapping one state in E to several states in E'. This is an important feature that is required to model scenarios such as the RailCab case study from [13]. The RailCab, described in more detail in the Validation Section, discusses a scenario in which the railway infrastructure has been enhanced with additional sensors. The current RailCab system receives messages for various milestones as it approaches a crossing (*endOfTrunkSection*, *lastBrake* and then *noReturn*. Its goals are stated in terms of actions it must perform between different milestones. The new system is to receive an additional message (*approachingCrossing*) for a newly introduced sensor that sits halfway between the *endOfTrunkSection* and the *lastBrake* milestones. Figure 4 (ignoring dashed transitions) depicts a portion of the environment model for the current (top) and new (bottom) RailCab specifications.

Non-determinism is introduced in the mapping from E to E' in the RailCab setting between the state in the current

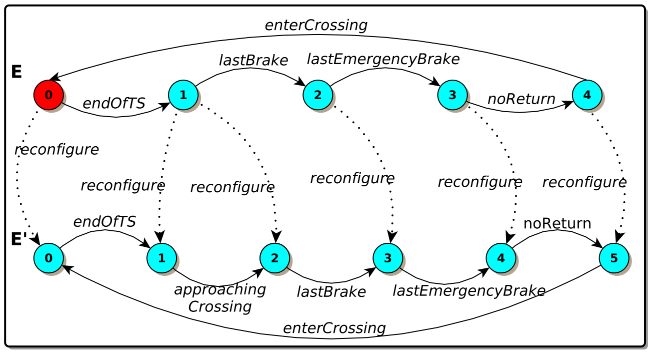


Fig. 4: Environment assumptions $(E \lightning_{R}^{reconfigure} E')$ for the Railcab dynamic update case study [13]. Dotted reconfigure transitions indicate how old assumptions are mapped into new ones according to relation R.

environment that models that the RailCab is between *endOfTrunkSection* and *lastBrake* and the two states in the new environment model that describe the fact that the RailCab is between *endOfTrunkSection* and *lastBrake*. Because the event *approachingCrossing* is not monitored in E, the correct way to model the transition to E' is to make no assumption on whether *approachingCrossing* has occurred. Indeed, Figure 4 describes the combination of both environments via an interrupt action $reconfigure$ and a relation ($R = \{(0, 0),$ $(1, 1), (1, 2), (2, 3), (3, 4), (4, 5)\}$).

On a methodological note, as with *hotSwap*, the alphabet of $(E \lightning_{R}^{reconfigure} E')$ is the union of the alphabets of E and E'. This entails that events in the communicating alphabet of E and not in that of E' (resp. in E' and not in E) are restricted from occurring after (resp. before) $reconfigure$.

4.3 Correctness Criteria for Dynamic Controller Update

We now formalise what it means for an dynamic controller update to be correct. In particular, what should the behaviour of a system be when a controller C that is guaranteeing a specification (in the form of current environment assumptions E, requirements $\Box G$ with G a propositional LTL formula, d a fluent definition, and controllable actions A) is replaced with a new controller that should satisfy a new specification (similarly, in the form E', $\Box G'$, d' and A') under a transition requirement T?

As an input to the correctness criteria, we require a specification of how the state of C' is to be set based on the state of C, (i.e. the relation H) and also how the state of the reconfigured environment E' is influenced by the state of E (i.e. the relation R).

We use the following term to model the behavior of updating C with C' such that the latter can change its environment E with E' via *reconfigure*: $(C \lightning_{H}^{hotSwap} C') \parallel (E \lightning_{R}^{reconfigure} E')$. $stopOldSpec$, $startNewSpec$ and $reconfigure$ should only be in the alphabet of C' thus stating that it is the new controller that will signal when the old specification is dropped, from when the new one is guaranteed and when the environment is to be reconfigured. As explained previously in this section, $hotSwap$ should not be in the alphabet of C, C', E, nor E'. H should be defined for each state of C. Also, $C \lightning_{H}^{hotSwap} C'$ should be legal for $E \lightning_{R}^{reconfigure} E'$ with respect to $(A_u, \widehat{A_u})$ where $A_u = A \cup A' \cup \{stopOldSpec, startNewSpec, reconfigure\}$ is the set of controllable actions for update, and, $\widehat{A_u} = \widehat{A} \cup \widehat{A'}$ is the set of uncontrollable actions for update. In other words, C and C' never block monitored events ($\ell \in \widehat{A_u}$) neither do controllable events that E and E' prohibit. In addition, note that *hotSwap* is an internal action of $C \lightning_{H}^{hotSwap} C'$ and we do not restrict the execution of this action (i.e. *hotSwap* $\notin A_u \cup \widehat{A_u}$).

In the following, we define the FLTL formula that models the expected behaviour of system $(C \lightning_{H}^{hotSwap} C') \parallel (E \lightning_{R}^{reconfigure} E')$:

Definition 4.1. (Goal for Dynamic Controller Update) *Let $\Box G$ and $\Box G'$ be the current and new goals for a system that is to go from a dynamic update of controllers, and G and G' are propositional FLTL formulae that do not include stopOldSpec, startNewSpec, reconfigure, nor hotSwap. Let T be an FLTL formula modelling the transition requirement that may refer to stopOldSpec, startNewSpec, and reconfigure, but not to hotSwap. We define G_u, the goal for a dynamic controller update as the conjunction of the following FLTL formulae:*

1) $G \ \mathbf{W} \ stopOldSpec$
2) T
3) $\Box(startNewSpec \implies \Box G')$
4) $\Box(hotSwap \implies (\Diamond stopOldSpec \wedge \Diamond reconfigure \wedge \Diamond startNewSpec))$

The first formula requires that the old goal G holds until the controller signals *stopOldSpec*. Recall that if the moment in which the old goal is dropped needs to be restricted, this must be specified in the transition requirement T. The second formula states that the transition requirement T must hold. It is expected to predicate over *stopOldSpec* and *startNewSpec* as it must constrain these from occurring based on what is considered a safe update state by the user. In addition, the third formula simply requires the new specification to hold from the point in which the controller signals *startNewSpec*. This will force the controller to only produce this signal only when it can ensure G'. The last formula requires the controller, once *hotSwap* occurs, to progress towards the occurrence of events *stopOldSpec*, *startNewSpec* and *reconfigure*.

The previous goal for the DCU problem must be evaluated with the fluent definition extension of d and d' (Definition 3.8). We will call it d_u. Note that the union of d and d' guarantees that the interpretation of G and G' does not change over E and E' respectively. The following definition puts all the parts together:

Definition 4.2. (Correctness Criteria for Dynamic Controller Update) *Let $\mathcal{P}$ be a tuple $(\mathcal{E}, C, \mathcal{E}', C', T, H, R, d_u)$ where, C and C' are LTS that model respectively the current controller and the new controller such that the communicating alphabet of C does not contain stopOldSpec, startNewSpec, reconfigure, nor hotSwap, and the communicating alphabet of C' contains stopOldSpec, startNewSpec, and reconfigure but not hotSwap; $\mathcal{E} = (E, \Box G, d, A, \widehat{A})$ and $\mathcal{E}' = (E', \Box G', d', A', \widehat{A'})$ are, respectively, the old and the new specification of systems for a dynamic controller update, where G and G' are propositional FLTL formulae that neither refer to stopOldSpec, startNewSpec,* *reconfigure, nor hotSwap; d_u is a fluent definition extension of d and d' as defined in Definition 3.8; T is a FLTL formula modelling the transition requirement, which may refer to stopOldSpec, startNewSpec, and reconfigure, but not to hotSwap; H and R are relations defining how C and E are interrupt to be replaced for C' and E' respectively; Let G_u be defined as in Definition 4.1;*

We say that $\mathcal{P}$ is a correct DCU if all the following hold:

(1) $(C \lightning_H^{hotSwap} C') \| (E \lightning_R^{reconfigure} E') \models_{d_u} G_u$.
(2) $(C \lightning_H^{hotSwap} C') \| (E \lightning_R^{reconfigure} E')$ is deadlock-free.
(3) Every state of C is mapped by H.
(4) $(C \lightning_H^{hotSwap} C')$ is a legal LTS for $(E \lightning_R^{reconfigure} E')$ with respect $(A_u, \widehat{A_u})$.

For the informal requirements for dynamic controller update discussed in Section 2, rules *ii)* through *v)* are captured in rules 1 through 4 of Definition 4.1. Rule *i)* is captured by the fact that *hotSwap* is not part of the alphabet of C, C', E and E', and H is defined for every rechable state in C. Thus, in the term $(C \lightning_H^{hotSwap} C') \| (E \lightning_R^{reconfigure} E')$, *hotSwap* is never constrained from occurring once.

In this paper we are interested in automatic approaches that produce correct solutions to dynamic controller update rather than a construct-then-verify approach. This can be formally described as follows:

Definition 4.3. (DCU Synthesis Problem) *Let C be a controller for the old specification $\mathcal{E}$, and $\mathcal{E}'$ be a new specification, T be a transition requirement, $R \subseteq (S_E \times S_{E'})$ be a relation, and d_u be a fluent definition extension of d and d'.*

A solution for the DCU Synthesis Problem is a pair (C', H) such that $(\mathcal{E}, C, \mathcal{E}', C', T, H, R, d_u)$ is a correct dynamic controller update.

Note that the DCU problem may or may not have a solution. The existence of a solution indicates that the controller resulting from solving the problem will guarantee G_u. The validity of the assumptions E and E' is the engineer's responsibility. In general, the non-existence of a solution arises due to too stringent combinations of specifications (E, G, E', G' and T) and lack of controllability by the controller of environment events.

The non-existence of a solution to the control problem can arise in three different situations. The first is that the new goal under the new environment assumptions are not achievable by a controller. For instance, there is no way to handle products to satisfy the new production process specification. In this case, the new specification must be corrected.

The second reason is that it is not possible to guarantee that the transition will satisfy T. For instance, requiring in T that the new specification be put in place immediately is impossible as products that have been partially processed may have been done so in a way that is inconsistent with the new specification. In this case, the transition requirement needs to be weakened.

Finally, non-existence of a solution may be due to the impossibility of guaranteeing that the update will *eventually* occur (even though it may actually occur). For instance, requiring the production line be emptied before changing cannot be guaranteed by the controller as it does not control the event *in(x)*. In this case the transition requirement needs to be changed to allow updates with a non-empty production line.

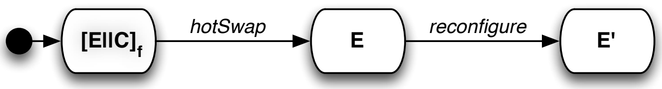


Fig. 5: Informal depiction of the three phases of environment E_u. First as an environment controlled by C but uncontrollable by C' ($[C \| E]_f$), then as an uncontrolled environment behaving as E and finally an uncontrolled environment behaving as E' resulting from the reconfigure operation.

5 A Proposed Solution

In this section we propose a solution to the dynamic controller update synthesis problem as formulated in Definition 4.3. The solution is based on recasting the DCU synthesis problem as an LTS control problem (Definition 3.10). Given a DCU synthesis problem (described by $\mathcal{E}$, C, $\mathcal{E}'$, T, R, and $\overline{d_u}$) we show how to build an LTS control problem $(E_u, G_u, \overline{d_u}, A_u)$ such that its solution C_u can be used to build a solution of the form (C', H) of a DCU synthesis problem.

We first show how to build E_u ($\overline{d_u}$ and A_u are straightforward) and then explain how to extract H and C' from C_u.

5.1 Environment Model

The environment E_u must model the reconfiguration of the environment from E to E' and also when the controller C_u must react to *hotSwap* event, or when it can use *stopOldSpec*, *startNewSpec* and *reconfigure* events. In addition, a key requirement for E_u is that it ensures that the resulting controller can be hot-swapped into the new system at any point, independently of the current state of the controller being swapped out.

In the following, we describe how to build E_u as an LTS with three phases. The first phase is designed to support hot-swapping, and ends when *hotSwap* occurs. The second models the behaviour of the current environment E, ending with *reconfigure*. The third models the behaviour of the new evnironment, E'. An informal representation of the three phases of E_u is depicted in Figure 5.

When C_u is constructed we need it to exhibit behavior equivalent to that of C up to the point where the hotswap occurs. We achieve this by first making E_u emulate the current system ($E_u \| C$) and ensuring that C_u, when computed, does not restrict its behaviour by *weakening* the control that C exerts: we make all events in $E_u \| C$ uncontrollable. Thus, in this first phase, C_u has no option but to simply monitor $E_u \| C$. (Recall the definition of LTS control, Definition 3.10 that the controller cannot constrain uncontrollable events). Consequently, the first phase of E_u may be defined as $[E \| C]_f$ where f is a relabelling function that maps all events $\ell \in A$ to fresh events that are not in $A_u = A \cup A' \cup \{stopOldSpec, startNewSpec, reconfigure\}$.

The second phase starts when *hotSwap* occurs. At this point the update controller must start restricting the behaviour of E to ensure a correct transition to satisfying

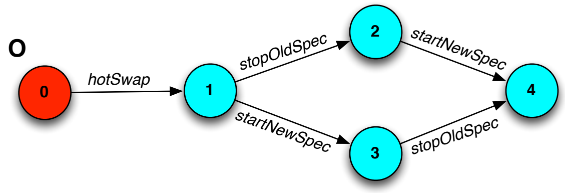


Fig. 6: LTS that define the occurrence of events *stopOldSpec* and *startNewSpec*. Before *hotSwap* action neither *stopOldSpec* nor *startNewSpec* can happen. After that, they can be signalled in any order.

the new specification, C is no longer active. Thus, the second phase can be described with E, while the transition between the first and second phase is modelled with the interrupt handler $\lightning$ on *hotSwap*. Then, the first two phases are captured by the term: $[E\|C]_f \lightning_I^{hotswap} E$, where I maps a state (e,c) of $[E\|C]_f$ to a state e of E.

Event *reconfigure* moves E_u from the second to the third phase, where the state of the new environment is described by the user providing the function R that maps states in E to states in E'. The three phases of E_u can therefore be described as $([E\|C]_f \lightning_I^{hotSwap} E \lightning_R^{reconfigure} E')$, as shown informally in Figure 5.

Finally, to simplify the specification of G, G' and T, and to allow a succinct rewrite of φ as a safety property G_u, we introduce in E_u events *stopOldSpec* and *startNewSpec*. These can be signalled once only in E_u after the *hotSwap* action and at any time before or after *reconfigure*. This is done by simply composing the LTS O depicted in Figure 6 with alphabet $\{hotSwap, stopOldSpec, startNewSpec\}$. As *reconfigure* is not in the alphabet of O we do not restrict its occurrence. In summary, the environment model E_u is defined as follows:

Definition 5.1. (Environment for the control problem) *Let C be the current controller, A be the set of events controlled by C, E and E' be the current and new environments, $R \subseteq (S_E \times S_{E'})$ be a relation provided by the user.*

The Environment for the DCU Synthesis Problem *is an LTS defined $E_u \triangleq ([E\|C]_f \lightning_I^{hotSwap} E \lightning_R^{reconfigure} E')\|O$ where I is a partial function such that $((e,c),e) \in I$ iff $(e,c) \in S_{E\|C}$, O is the LTS depicted in figure 6 with alphabet $\{hotSwap, stopOldSpec, startNewSpec\}$, and f is the function that relabels all events $\ell \in A$ to fresh events $\overline{\ell} \notin A$.*

5.2 Solving DCU synthesis with LTS control

To derive a correct solution to the DCU synthesis problem, we first define and solve an LTS control problem (Definition 3.10) and then extract a controller C' and a relation H.

A solution to an LTS control problem can be constructed using the environment E_u as defined in Definition 5.1, the goal G_u as defined in Definition 4.1, the set of controlled events A_u defined as the union of the controlled events A, A' and {*stopOldSpec*, *startNewSpec*, *reconfigure*} and the set of uncontrolled events $\widehat{A_u}$ defined as the union of the uncontrolled events $\widehat{A}$, $\widehat{A}'$. To produce a controller that does not restrict the execution of *hotSwap*, we will include this action in this set. Lines 2 to 6 in Algorithm 1 show how to generate these elements.

ALGORITHM 1: Pseudocode for extracting the solution to DCU synthesis problem $\mathcal{S}$ from solving the LTS control problem with specification $\mathcal{E}_u$.

```
1  DynamicUpdateofControllers ($\mathcal{E}$, $C$, $\mathcal{E}'$, $T$, $R$, $d_u$, $f$)
2      $E_u \leftarrow$ buildEnv($\mathcal{E}$, $C$, $\mathcal{E}'$, $R$, $f$); // Definition 5.1
3      $G_u \leftarrow$ buildGoal($\mathcal{E}$, $T$, $\mathcal{E}'$); // Definition 4.1
4      $A_u \leftarrow$ {stopOldSpec, startNewSpec, reconfigure };
5      $A_u \leftarrow A_u \cup \mathcal{E}.A \cup \mathcal{E}'.A$;
6      $\widehat{A_u} \leftarrow \mathcal{E}.\widehat{A} \cup \mathcal{E}'.\widehat{A}$;
7      $\overline{d_u} \leftarrow \emptyset$;
8      foreach $\langle I, F, Init \rangle \in d_u$ do
9          $\overline{I} \leftarrow I \cup \{f(\ell) \mid \ell \in I\}$;
10         $\overline{F} \leftarrow F \cup \{f(\ell) \mid \ell \in F\}$;
11         $\overline{d_u} \leftarrow \overline{d_u} \cup \{\langle \overline{I}, \overline{F}, Init \rangle\}$
12     end
13     $\mathcal{E}_u \leftarrow (E_u, G_u, \overline{d_u}, A_u, \widehat{A_u} \cup \{hotSwap\})$;
14     $C_u \leftarrow$ solve($\mathcal{E}_u$); // Definition 3.10
15     if $C_u$ is null then
16         return DCU has no solution
17     $H \leftarrow \emptyset$;
18     foreach $c \in states(C)$ do
19         $q \leftarrow$ getBisimilarStateUpToHotSwap($c$, $C_u$);
20         $c'_u \leftarrow$ getSuccessor($q$, hotSwap, $C_u$);
21         $H \leftarrow H \cup \{(c, c'_u)\}$;
22     end
23     $C'_u \leftarrow$ removeStatesBeforeHotSwap($C_u$);
24     return ($C'_u$, $H$)
```

Note that traces in C guaranteeing G will not satisfy G_u when relabelled by f in E_u (recall f from Definition 5.1). To make traces in $[E\|C]_f$ be accepted by G_u, it is necessary to modify slightly the fluent definition extension d_u. We use a fluent definition extension $\overline{d_u}$ where each fluent definition in d_u is changed as follows: if a fluent is defined as $\langle I, T, Init \rangle$ in d_u, then, it will be defined as $\langle I \cup \{f(\ell) \mid \ell \in I\}, F \cup \{f(\ell) \mid \ell \in F\}, Init \rangle$ in $\overline{d_u}$. This construction of $\overline{d_u}$ from d_u generates a fluent definition extension of d and d' because for each fluent definition in d_u we add events to its initiating and terminating sets that are not in the alphabet of d nor d'. Thus, fluent definition extension conditions in Definition 3.8 hold. The first foreach in Algorithm 1 (lines 7 to 12) shows how to build $\overline{d_u}$ from d_u.

If there exists a solution C_u to the DCU synthesis problem with specification $(E_u, G_u, \overline{d_u}, A_u, \widehat{A_u} \cup \{hotSwap\})$, it can be shown to have a set of states, all of them reachable from the initial state, that are bisimilar to $[C]_f$ up to the occurrence of $hotSwap$ (see Section 5.4). Indeed, C_u will have three phases, the first in which is bisimilar to $[C]_f$ (see Section 5.4), the second in which it is handling the transition between specifications (dealing with $\Box G$, $\Box G'$, T and reconfiguration) and the third in which both *stopOldSpec* and *startNewSpec* have occurred and C_u is simply dedicated to satisfying $\Box G'$ (see Figure 7).

The first phase allows the definition of H from states of C to those of C_u: as $[C]_f$ is bisimilar to C_u up to *hotSwap*, then, for each $c \in S_C$ there is at least one state $q \in S_{C_u}$ such that c and q are bisimilar. In addition, transition $(q, hotSwap, c'_u)$ must exist in C_u and be unique because

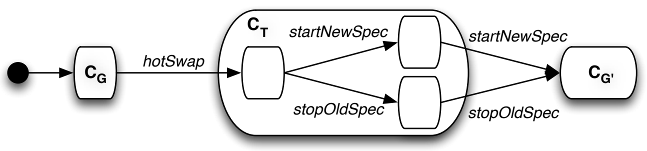


Fig. 7: Informal depiction of the three phases of controller C_u. First behaving exactly like C and thus guaranteeing $\Box G$ (C_G), then controlling the transition period (C_T) and once both *stopOldSpec* and *startNewSpec* have occurred, guaranteeing $\Box G'$ ($C_{G'}$). Note that, for simplicity, *reconfigure* is not depicted.

every state up to *hotSwap* has this uncontrollable event enabled in E_u. H should map c to c'_u. Lines from 17 to 22 in Algorithm 1 shows a pseudocode for building H where the method called **getBisimilarStateUpToHotSwap** returns only one state q and **getSuccessor** returns the reachable state after transitioning through *hotSwap* from q.

Having defined H, we now construct C'_u from C_u. Note that H maps all states in C to states in the second phase of C_u (see C_T in Figure 7). This means that states in the first phase of C_u will never be visited in $C \lightning^{hotSwap}_{H} C_u$. Thus, we construct C'_u to be the portion of C_u that does not include states from its first phase. This ensures that C'_u does not have a complete replica of C within it, thereby avoiding bloating of the controller through successive updates. The method called **removeStatesUpToHotSwap** at Line 23 in Algorithm 1 performs the procedure described in this paragraph.

5.3 Complexity

Solving an LTS control problem (as defined in Definition 3.10 in Section 3) for an arbitrary FLTL property is 2EXPTIME complete [25]. It is straightforward to show that if T is a safety property then G_u can be encoded as an obligation property (i.e. disjunction of safety and reachability assertions, $\bigwedge_{i=1}^{n}(\Box S_i \vee \Diamond R_i)$). LTS control problems with goals in the form of obligations can be resolved in linear time with respect to the size of E_u for deterministic environment models. For non-deterministic environments, a specialised sub-set construction can be used to produce a deterministic version, however an exponential explosion can occur depending on the degree of non-determinism [26]. The same price is paid for allowing partial observability (in this paper we assume all events not controlled are observable to the controller) can similarly be reduced to a deterministic problem with the same cost.

We have extended our synthesis tool [27] to support the specification of a DCU problem with safety transition properties (T), the automatic construction of an LTS control problem as in Definition 3.10, the resolution of this problem and the extraction of a solution (C'_u, H) to the specified DCU problem. The case studies described below were solved using this tool. Both case studies and tool are available at [28].

5.4 Soundness and Completeness

In the following we state and prove the soundness and completeness of our approach. By soundness we mean that the Algorithm described in Section 5.2 indeed produces a correct solution to the DCU synthesis problem. By completeness we mean that if the algorithm does not propose a solution then there is no solution to the DCU problem.

Theorem 5.1. *Let $\mathcal{S}$ be a DCU synthesis problem with specification $(\mathcal{E}, C, \mathcal{E}', T, R, d_u)$, E_u be defined as in Definition 5.1 using function f for relabelling, G_u be defined as in Definition 4.1, A_u be the union of old and new controllable actions plus special events stopOldSpec, startNewSpec, and reconfigure, and $\widehat{A_u}$ be the union of old and new uncontrollable actions.*

[*Correctness*] *if C_u is the solution to the LTS control problem with specification $\mathcal{E}_u = (E_u, G_u, \overline{d_u}, A_u, \widehat{A_u} \cup \{hotSwap\})$, then, building (C'_u, H) as in Algorithm 1 is a solution of $\mathcal{S}$.*

[*Completeness*] *if $\mathcal{S}$ has a solution, Algorithm 1 returns a solution.*

For the following proof we use $\lightning^{hs}$ and $\lightning^{rec}$ as a shorthand for $\lightning^{hotSwap}$ and $\lightning^{reconfigure}$ respectively.

Correctness Proof. We must prove that Algorithm 1 returns (C'_u, H) solution to $\mathcal{S}$. From lines 17 to 23 in Algorithm 1, C'_u and H are defined as follows: $C_u = ([C_u]_{cut} \lightning^{hs}_{Q} C'_u)$ and $(c, c'_u) \in H$ iff $\exists q \cdot (q, c'_u) \in Q$ and $[C_u]_{cut}(q) \sim [C]_f(c)$ with $cut(\ell) = \ell$ iff $\ell \neq hotSwap$ (see Figure 8 for a better understanding). Thus, by definition 4.2 we must prove that:

(1) $(C \lightning^{hs}_{H} C'_u) \parallel (E \lightning^{rec}_{R} E') \models_{d_u} G_u$.
(2) $(C \lightning^{hs}_{H} C'_u) \parallel (E \lightning^{rec}_{R} E')$ is deadlock-free
(3) Every state of C is mapped by H.
(4) $(C \lightning^{hs}_{H} C'_u)$ is a legal LTS for $(E \lightning^{rec}_{R} E')$ with respect to $(A_u, \widehat{A_u})$.

For items (1) and (2) we first show that

$$C_u \| E_u \sim ([C]_f \lightning^{hs}_{H} C'_u) \ \|([E]_f \lightning^{hs}_{Id} E \lightning^{rec}_{R} E').$$

Then, using that $C_u \| E_u \models_{\overline{d_u}} G_u$ and $C_u \| E_u$ is deadlock-free we prove items (1) and (2) by contradiction

We first need to show that C_u can always be decomposed into $([C_u]_{cut} \lightning^{hs}_{Q} C'_u)$: the decomposition is possible because *hotSwap* is enabled in all states of C_u until *hotSwap* occurs for the first time (see Lemma 5.1). Also, it is straightforward to show that $[C_u]_{cut} \sim [C]_f$ (see Lemma 5.2). Thus:

$$C_u \sim ([C]_f \lightning^{hs}_{H} C'_u)$$

where by replacing $[C_u]_{cut}$ for $[C]_f$ we need to change Q for H (see Figure 8).

Now, we will work with the environment E_u. By Definition 5.1, $E_u \triangleq ([E \| C]_f \lightning^{hs}_{I} E \lightning^{rec}_{R} E') \| O$, and as E and C are deterministic LTS, by Property 3.1:

$$E_u \sim (([E]_f \| [C]_f) \lightning^{hs}_{I} E \lightning^{rec}_{R} E') \ \| \ O$$

Note that O is an observer that restricts the occurrence of *stopOldSpec* and *startNewSpec* before *hotSwap* and only allows each of them to occur once. C_u already exhibits this behaviour as it is a solution for control problem $\mathcal{E}_u$. Being both C_u and O deterministic, $C_u \| O$ is bisimilar to C_u, and therefore:

$$C_u \| E_u \sim ([C]_f \lightning^{hs}_{H} C'_u) \parallel (([E]_f \| [C]_f) \lightning^{hs}_{I} E \lightning^{rec}_{R} E')$$

By Property 3.2 we have:

$$C_u \| E_u \sim (([C]_f \| [E]_f \| [C]_f) \lightning^{hs}_{H'} C'_u) \ \parallel \\ (([C]_f \| [E]_f \| [C]_f) \lightning^{hs}_{I'} E \lightning^{rec}_{R} E')$$

As $[C]_f$ is deterministic we have $[C]_f \| [C]_f \sim [C]_f$, thus:

$$C_u \| E_u \sim \\ (([C]_f \| [E]_f) \lightning^{hs}_{H'} C'_u) \ \parallel \ (([C]_f \| [E]_f) \lightning^{hs}_{I'} E \lightning^{rec}_{R} E')$$

Applying Property 3.2, and by definition of I in Definition 5.1, we have:

$$C_u \| E_u \sim ([C]_f \lightning^{hs}_{H} C'_u) \ \|([E]_f \lightning^{hs}_{Id} E \lightning^{rec}_{R} E')$$

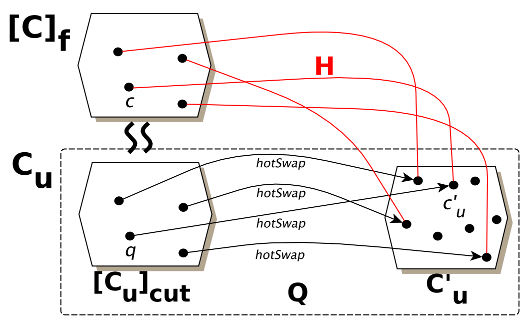


Fig. 8: Support for proof sketch. Informal depiction of $C_u = ([C_u]_{cut} \lightning^{hs}_{Q} C'_u)$. We show that for given states c and q such that $[C_u]_{cut}(q) \sim [C]_f(c)$, then, H maps c to state c'_u in C'_u according to relation Q.

Now we start proving item (1) by contradiction.

Let assume $(C \lightning^{hs}_{H} C'_u) \parallel (E \lightning^{rec}_{R} E') \not\models_{d_u} G_u$.

By Lemma 5.3 we know that $(C \lightning^{hs}_{H} C'_u) \parallel (E \lightning^{rec}_{R} E')$ is bisimilar to $(C \lightning^{hs}_{H} C'_u) \parallel (E \lightning^{hs}_{Id} E \lightning^{rec}_{R} E')$, and therefore , $(C \lightning^{hs}_{H} C'_u) \parallel (E \lightning^{hs}_{Id} E \lightning^{rec}_{R} E') \not\models_{d_u} G_u$.

Now, consider a trace $\pi \in (C \lightning^{hs}_{H} C'_u) \parallel (E \lightning^{hs}_{Id} E \lightning^{rec}_{R} E')$ such that $\pi \not\models_{d_u} G_u$. Note that, π is either a trace such that $\pi = \ell_0, \dots, \ell_n, \dots$ and $hotSwap \notin \pi$, or $\pi = \ell_0, \dots, \ell_i, hotSwap, \ell_{i+1}, \dots$.

We can construct a trace $\overline{\pi} \in ([C]_f \lightning^{hs}_{H} C'_u) \parallel ([E]_f \lightning^{hs}_{Id} E \lightning^{rec}_{R} E')$ such that either $\overline{\pi} = f(\ell_0), \dots, f(\ell_n), \dots$ and $hotSwap \notin \overline{\pi}$, or $\overline{\pi} = f(\ell_0), \dots, \underline{f(\ell_i)}, hotSwap, \ell_{i+1}, \dots$. Furthermore, $\overline{\pi} \not\models_{\overline{d_u}} G_u$ because $\overline{d_u}$ changes a value of a fluent for $f(\ell)$ if and only if d_u does so for ℓ.

Trace $\overline{\pi}$ introduces a contradiction from assuming that $(C \lightning^{hs}_{H} C'_u) \| (E \lightning^{rec}_{R} E') \not\models_{d_u} G_u$.

Proving item (2), i.e., that $(C \lightning^{hs}_{H} C'_u) \parallel (E \lightning^{rec}_{R} E')$ is deadlock free follows a similar reasoning.

To prove item (3) we need to find for each $c \in S_C$, a state c'_u in C'_u mapped by H. Thus, we need to prove that there exists a state $q \in S_{[C_u]_{cut}}$ such that $(q, c'_u) \in Q$ and $[C_u]_{cut}(q) \sim [C]_f(c)$ (see Figure 8).

Let π be a finite trace in $[C]_f$ that reaches a state c. Then, π can also be executed in $[C_u]_{cut}$ reaching a state q because $[C_u]_{cut} \sim [C]_f$ (see Lemma 5.2), and therefore, $[C_u]_{cut}(q) \sim [C]_f(c)$. What remains to be shown is that there is a c'_u such that $(q, c'_u) \in Q$, in other words that in C_u there is a *hotSwap* transition from q to c'_u. This is straightforward: as $q \in [C_u]_{cut}$ and $C_u \sim ([C_u]_{cut} \lightning^{hs}_{Q} C'_u)$ with Q a total function (Lemma 5.1), by definition of $\lightning$ that transition must exist.

To prove item (4), we know that for any trace π in $(C \lightning^{hs}_{H} C'_u) \parallel (E \lightning^{rec}_{R} E')$, that reaches a state (c, e), the following holds.

If $hotSwap \notin \pi$ then $c \in S_C$ and $e \in S_E$. Furthermore, (c, e) is reachable in $C \| E$ via π. As C is a solution to $\mathcal{E}$, then C is a legal LTS for E with respect to $(A, \widehat{A})$ and c and e are legal with respect to $(A, \widehat{A})$. This entails that they are also legal with respect to $(A_u, \widehat{A_u})$, because C and E have no transitions labelled with $A_u \setminus A$ or $\widehat{A_u} \setminus \widehat{A}$.

If $hotSwap \in \pi$ then we can build $\overline{\pi}$ that reaches $(c, e) \in C_u \| E_u$ by simply relabelling every ℓ appearing in π before *hotSwap* with $f(\ell)$. As C_u is a legal LTS for E_u, c and e are legal with respect to $(A_u, \widehat{A_u} \cup \{hotSwap\})$. As there is no *hotSwap* transition enabled from these states then c and e are also legal with respect to $(A_u, \widehat{A_u})$. □

Completeness Proof. Assuming a solution (C', Y) to $\mathcal{S}$, we must prove the existence of an LTS X that is a solution for the LTS control problem $\mathcal{E}_u$. Let X be such that $X \triangleq ([E\|C]_f \lightning^{hs}_{Y'} C')$ where $((e, c), c') \in Y'$ iff $(c, c') \in Y$. In other words that:

(a) $X \| E_u \models_{\overline{d_u}} G_u$.
(b) $X \| E_u$ is deadlock-free, and
(c) X is a legal LTS for E_u with respect to $(A_u, \widehat{A_u} \cup \{hotSwap\})$,

We first prove item (a):

As (C', Y) is a solution to $\mathcal{S}$ we know that

$$(C \lightning^{hs}_{Y} C') \parallel (E \lightning^{rec}_{R} E') \models_{d_u} G_u,$$

where Y maps every state of C to C'.

We use as before Lemma 5.3 to decompose E:

$$(C \lightning^{hs}_{Y} C') \parallel (E \lightning^{hs}_{Id} E \lightning^{rec}_{R} E') \models_{d_u} G_u.$$

where Id is the identity relation.

Moreover, using Property 3.2, we have:

$$(E\|C \lightning^{hs}_{Y'} C') \parallel (E\|C \lightning^{hs}_{I} E \lightning^{rec}_{R} E') \models_{d_u} G_u.$$

where $((e, c), e) \in I$ iff $(e, e) \in Id$.

We can relabel controllable actions in $E\|C$ with f while still $\underline{\text{guaranteeing}}$ G_u if we change the fluent definition d_u with $\overline{d_u}$. The following is entailed:

$$([E\|C]_f \lightning^{hs}_{Y'} C') \parallel ([E\|C]_f \lightning^{hs}_{I} E \lightning^{rec}_{R} E') \models_{\overline{d_u}} G_u.$$

As before, we have that $([E\|C]_f \lightning^{hs}_{Y'} C') \| O$ is bisimilar to $([E\|C]_f \lightning^{hs}_{Y'} C')$ where O is the observer introduced in Definition 5.1. This is because O prohibits to do *stopOldSpec* and *startNewSpec* before *hotSwap*, and allows each one to occur at most once. Thus we have:

$$([E\|C]_f \lightning^{hs}_{Y'} C') \parallel O \parallel ([E\|C]_f \lightning^{hs}_{I} E \lightning^{rec}_{R} E') \models_{\overline{d_u}} G_u.$$

Finally, using definition of E_u item (a) is proved:

$$([E\|C]_f \lightning^{hs}_{Y'} C') \parallel E_u \models_{\overline{d_u}} G_u$$

The proof of item (b) follows the same reasoning as above. Starting from the fact that $(C \lightning^{hs}_{Y} C') \parallel (E \lightning^{hs}_{Id} E \lightning^{rec}_{R} E')$ is deadlock free and via the same bisimulation preserving transformations we reach that $([E\|C]_f \lightning^{hs}_{Y'} C') \parallel E_u$ is also deadlock free.

We prove item (c) in straightforward manner. For any trace $\overline{\pi}$ in $X \| E_u$ that reaches a state (x, e), we know the following:

If $hotSwap \notin \overline{\pi}$ then $x \in [E\|C]_f$, and, by definition of E_u, we also have $e \in [E\|C]_f$. As $[E\|C]_f$ is a deterministic LTS, then, $x = e$. Hence, we have that $\Delta_X(x) = \Delta_{E_u}(e)$. If so, x and e must be legal with respect to any pair of sets.

If $hotSwap \in \overline{\pi}$ then we can build π that reaches the same pair (x, e) but in $(C \lightning^{hs}_{Y} C') \| ((E\|C) \lightning^{hs}_{I} E \lightning^{rec}_{R} E')$ by simply replacing every relabelled event $f(\ell)$ appearing in $\overline{\pi}$ before *hotSwap* for ℓ. Furthermore, π up to *hotSwap* leads in $(E\|C) \lightning^{hs}_{I} E$ to the same state in E as when π is run directly on E. Thus, π leads to (x, e) in $(C \lightning^{hs}_{Y} C') \parallel (E \lightning^{rec}_{R} E')$. As $(C \lightning^{hs}_{Y} C')$ is a legal LTS for $(E \lightning^{rec}_{R} E')$, x and e are legal with respect to $(A_u, \widehat{A_u})$. As there is no *hotSwap* transition enabled from these states, then, x and e are also legal with respect to $(A_u, \widehat{A_u} \cup \{hotSwap\})$. □

Proofs for the following lemmas, used in the proof above, are straightforward.

Lemma 5.1. *Let C_u be a solution to the LTS control problem with specification $\mathcal{E}_u$, then, there exists C'_u such that $C_u \sim ([C_u]_{cut} \lightning_{Q}^{hotSwap} C'_u)$ with Q a total function.*

Lemma 5.2. *Let C_u be a result of an LTS control problem with specification $\mathcal{E}_u$, then, $[C_u]_{cut} \sim [E \| C]_f \sim [C]_f$.*

Lemma 5.3. *Let $(C \lightning_{H}^{hs} C'_u) \parallel (E \lightning_{Id}^{hs} E \lightning_{R}^{rec} E')$ and $(C \lightning_{H}^{hs} C'_u) \parallel (E \lightning_{R}^{rec} E')$ be LTS, then, they are bisimilar.*

In this section we have presented a sound and complete approach, with linear time complexity when transitions requirements are safety properties, to solving DCU synthesis problems by converting it to an LTS control problem.

6 Validation

In this section we report on the case studies we have run to validate our approach. The purpose of our validation is to show applicability and generality of the approach with respect to existing work using case studies taken from literature and from industry. As discussed, the complexity of the DCU synthesis problem is linear and hence scalability is not a primary concern.

All case studies were run using an extension of the MTSA tool [27], an extension of LTSA[29]. Like LTSA, MTSA natively supports specification of LTS [30] and properties using a textual process algebra and FLTL [21]. The tool also supports synthesis of controllers for SGR(1) control problems, which are strictly more expressive than is required for DCU control problems. The tool was extended to support definition of DCU control problems, computing A_u, G_u and E_u, and solving the DCU control problems. The extended version of the tool and case studies can be found at [28].

The selected case studies are the Rail Cab [13], Power Plant [14], GSM-oriented protocol [12] and MetaSocket [11] to allow comparison with the work closest to ours. A dynamic workflow update case from [31] and the Production Cell [17] were included to illustrate how the current limitation of workflow systems technology could be overcome (currently limited to requiring quiescence before updating, e.g., [32]). We also used a workflow system case study provided by an industrial partner. Finally, we chose a UAV surveillance setting inspired from [33] aimed showing an end-to-end application of the technique, from synthesis to enactment on an adaptive system infrastructure.

For each case study we experimented with a number of different situations. We fixed the old and new specifications for the update and explored the use of various transition requirements. In addition to domain specific transition requirements defined for each case study, we used two default transition requirements $T_\top$ and $T_\emptyset$ on all case studies.

$T_\top = \top$ is a non restricting transition requirement which permits the old specification to be dropped at any point and also a period in which anything is allowed before starting the new specification. As the controllers we construct are eager to achieve liveness (i.e., to start enforcing the new specification), the period in which anything goes will be minimal in the sense that the controller will do just enough to reach a state from which the new specification can be enforced. The second default requirement used prohibits events from occurring during the period in which neither specification holds ($T_\emptyset = (\neg \mathit{NewSpecStarted} \wedge \mathit{OldSpecStopped}) \Rightarrow \neg \mathit{Event}$, where Event is a disjunction of all events).

For every case study and transition requirement we also built a controller that is safe but that does not assure progress towards the specification change. The purpose of this was to be able to compare the safe and live controller against the safe but not live one. This permitted us to ensure that the liveness requirement does indeed induce an update controller that guides the system to a safe state in which the transition requirements can be satisfied and that the new specification can be achieved. Additionally, producing both controllers gives an indication of the additional computational cost of solving a live dynamic controller update problem. We provide a qualitative discussion on the differences between the live and non-live controllers for the first case study, the Production Cell. For the rest, we show only quantitative information: computational cost and controller size (see Table 1).

6.1 Production Cell

The general setting for the production cell discussed in Section 2, which is based on [17], is that of industrial automation in which a production line being controlled to process products according to one specification must be updated to accommodate new production rules. For this, we modelled various alternative transition requirements and built update controllers for each of them.

The first transition requirement was one in which the new specification is required to be put in place when the production line has no products being processed ($\mathit{startNewSpec} \implies \mathit{Empty}$). As expected, our tool reported that there is no update controller that can satisfy this requirement. This is reasonable as *in(x)* is not an action that the controller can control: If new products arrive sufficiently regularly, the production line will never be empty and thus the plant will never be in a state where the new production rules can come into effect.

A weaker transition requirement is to force the plan to always satisfy one of the specifications ($T_\emptyset$). The result is a controller that delays switching to the new production rules until all elements that are on the production line are raw (i.e. no tool has been applied to them). This is because in this particular case, rules for processing products of the two specifications are incompatible. If a product has started to be processed using the old rules, then it is impossible to comply to the new ones. The only strategy for a partially processed product is to finish producing it and to delay processing any new product that may be put onto the production line (*in(x)*).

In Figure 9 we depict the update controller for $T_\emptyset$. As explained in Section 5.2, the dotted box on the left C_G contains the states of the update controller that are structurally equivalent to controlling the production according to the old production rules. The box on the right $C_{G'}$ contains all states of the update controller that describe the behaviour of the controller once it has finished the transition period between specifications and guarantees the new production rules. The behaviour described between the two boxes describes the strategy of the update controller to reach a state in which it can effectively switch specifications. Note that the behaviour between the boxes has no loops, and thus guarantees progress towards update.

In contrast, and to emphasise the importance of assuring progress in dynamic controller updates, Figure 10 shows the result of building a controller with the transition requirement $T_\emptyset$ without the liveness goal requirement for dynamic controller update (Definition 4.1). The safe but not live update controller allows looping behaviour once it has been hotswapped. This loop, depicted in red, permits the controller to continue to fully process any new raw product that enters the production line according to the old specification (*in, drill, drillOk, polish, polishOk, out*), thus never starting to process products using the new specification.

As discussed in Section 2, we also studied a transition requirement (T_1) which specifies that the switch in production rules should occur when all products on the production line have not been polished. To achieve this change, the update controller must continue to process products that have been polished but must halt processing products that have not reached the polish phase so as to ensure that eventually a state is reached in which T_1 holds. Removing the liveness requirement yields a much larger controller (537 states against 107) that does not guarantee the update will actually happen.

Finally, we computed the update controller for T_2 (see Section 2) that allows a period in which neither the old nor the new production specification holds but that requires that during this intermediate period products are either processed according to the new specification or are trashed. The safe and live controller is roughly a quarter of the size of the safe but not live controller.

6.2 Power Plant

In [14] a controller for the cooling system of a nuclear power plant is discussed. In order to service maintenance requests, the existing controller first stops the cooling agent pump and then restarts it. In a new controller specification, it is no longer required to stop and restart the pump, and there is a system invariant which states that the cooling agent pump may not be stopped indefinitely. The authors show that if an update is performed naively at a state in which the current controller has stopped the pump but not yet restarted it, then the plant risks a dangerous incident as the new controller may not restart the pump. A safe way to satisfy the system invariant, they argue, is to require dynamic update to preserve the behaviour of an offline update (i.e. equivalent to updating when it has restarted the pump).

We considered three different DCU control problems for this case study. Both with $T_\top$ and $T_\emptyset$ the system exhibits the invalid behaviour described in [14]. We also used $T = T_\emptyset \wedge (startNewSpec \Rightarrow PumpOn)$ to require that in the transition period, should the pump be off, the pump must be started before the end of the next maintenance procedure. This requirement avoids leaving the pump off unintentionally. Furthermore, it is less restrictive than the "equivalent to offline update" requirement in [13, 14]. Not only does T allow specification updates in strictly more states than in [13] while satisfying the desireable requirement ($\neg\Diamond\Box PumpOff$), it also avoids the manual correctness validation required in [14] for weaker transition requirements than [13].

6.3 RailCab

The RailCab system [34] consists of autonomous vehicles that coordinate the transport of passengers and goods on demand. The subsystem discussed in [13] focuses on control of RailCabs as they approach a crossing. The RailCab can monitor events such as *endOfTrunkSection* and that it has passed the *lastBrake* or *lastEmergencyBrake* milestones. It controls the *brake* and *emergencyBrake* and also can receive responses to queries it controls such as *requestEnter*. The goals for the current and new controller are required to ensure that the RailCab enters the crossing only if it has been granted permission to do so. There are also constraints on when permission may be requested and when brakes can be applied, and assumptions on when responses to controlled actions occur. The monitorable events such as milestones and responses to the system are depicted in Figure 11 with black and red fonts respectively. Controllable actions are shown in blue.

The difference between the specifications for the current and the new controller is that the new specification introduces a new monitored event and additional goals. The new controller monitors event *approachingCrossing* which is assumed to be received at a milestone that occurs between the *endOfTrunkSection* and *lastbrake*. The new controller is required to perform an additional *checkCrossing* on the crossing barrier status between the *approachingCrossing* and the *lastbrake*.

The difficulty with this change of specification is what to do when the system is to be updated having passed *endOfTrunkSection* but not yet reached *lastbrake* (see Figure 11). In this scenario, the current system may have unknowingly passed the *approachingCrossing* milestone because it cannot monitor the *approachingCrossing* event. If the system were updated between these milestones, the updated system would not know whether it must start with the additional checks that are required. Doing so without having passed *approachingCrossing*, would violate the intent of the new specification (i.e. that *checkCrossing* must occur after passing *approachingCrossing*).

In [13] the problem is resolved by simply not allowing the system to be updated if an update is requested while the RailCab has received *endOfTrunkSection*. In this case, the update is postponed until after the RailCab has passed the crossing. This postponement is unsatisfactory as a security update for a critical system should not be postponed unless strictly necessary.

Using the approach described in this paper we can update the system even during the period in which the RailCab is in the *endOfTrunkSection-lastbrake* zone. The resulting controller update will *reconfigure* the system to enable the monitoring of *approachingCrossing* events but will have some uncertainty regarding whether event *approachingCrossing* is received before *lastbrake*. The key is that if the update controller then receives *approachingCrossing*, it can go on to satisfy the new specification by performing *checkCrossing*. On the other hand, if the update controller receives *lastbrake* it has no choice but to continue with the old specification resulting in behaviour similar to [13]

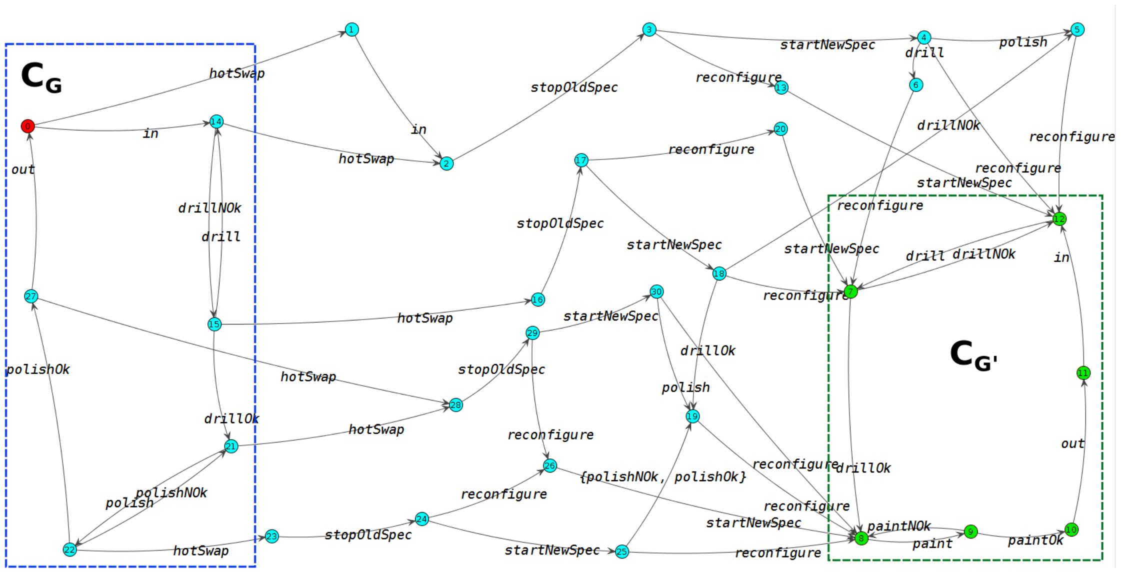


Fig. 9: Update controller for a reduced (two tool) version of the Production Cell with its three phases. C_G and $C_{G'}$ guarantee the old and new specifications. The middle portion ensures progress between both while guaranteeing $T_\emptyset$.

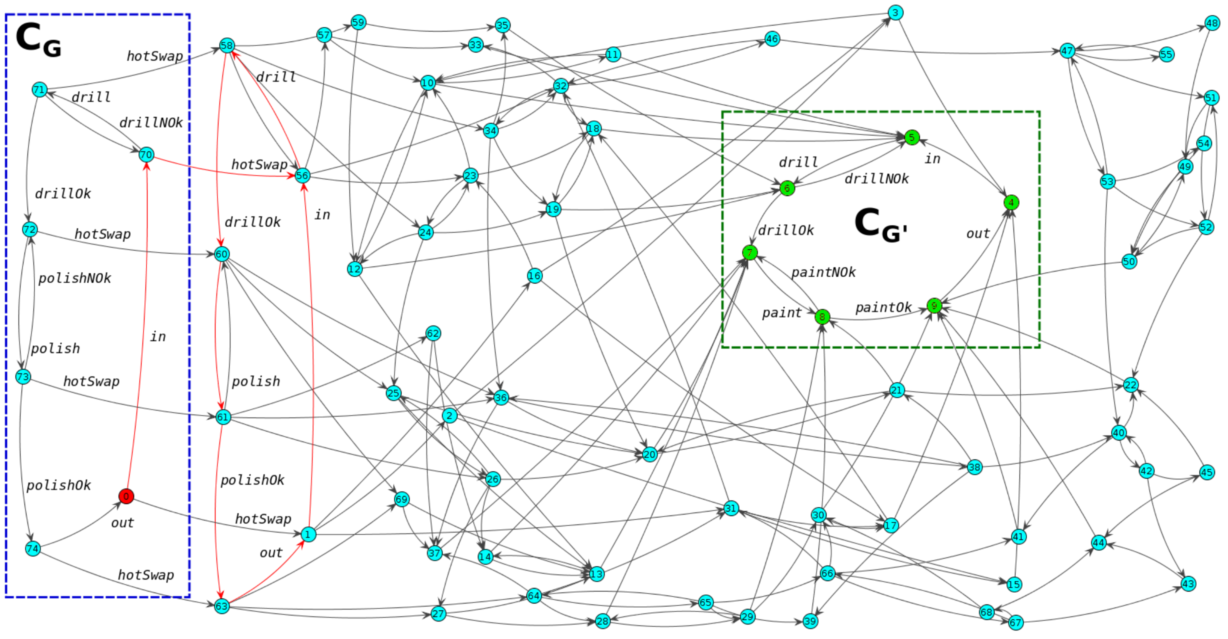


Fig. 10: Safe but not live Controller C_u^S for a reduced (two tool) version of the Production Cell with its three phases. In contrast to Figure 9, the middle phase has loops (e.g., transitions in red) which do not guarantee update completion.

To produce the update controller described above, we make use of a non-deterministic mapping between states of the old and new environment models. That is, the relation R must map the state of E that corresponds to the *endOfTrunkSection-lastbrake* zone to two states in E, the one modelling that the RailCab is in the *endOfTrunkSection-approachingCrossing* zone and the one for the *approachingCrossing-lastbrake* zone (see Figure 4).

We resolved various DCU control problems for three different transition requirements. For $T_\top$, the controller exhibits unsafe behaviour. Using $T_\emptyset$ the resulting controller can perform a safe update in more states than the one in [13]: if an update is requested while the RailCab is between *endOfTrunkSection* and *lastBrake*. The problem with $T_\emptyset$ is that it allows the update controller to try *approachingCrossing* after *reconfigure* but before *startNewSpec* which in practice leads to a violation of the new specification even if the update controller has not committed to satisfying it yet. Hence, we use $T = T_\emptyset \wedge (\neg OldSpecStopped \Rightarrow \neg CheckCrossing)$ as one possible comprehensive specification for the RailCab update problem.

6.4 GSM-oriented protocol

In [12] a case study is presented based on a GSM-oriented protocol [12] used to transfer audio stream in a lossy wireless network. The authors describe an update between two

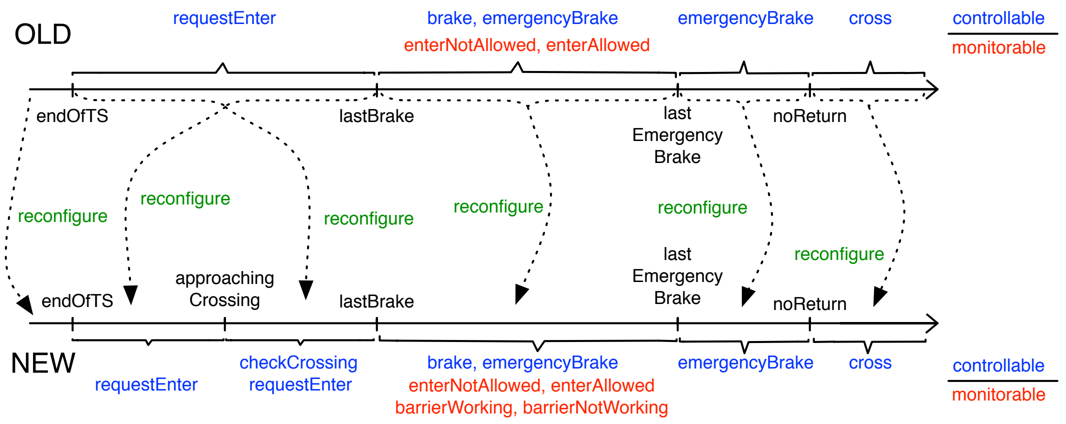


Fig. 11: Railcab timeline. Milestones in black font, controllable events in blue font and monitorable events in red font.

implementations of the GSM protocol which use different piggybacking strategies to enhance data transfer. Increasing the amount of piggybacking supports the handling of networks with higher loss rates. Consequently, performing dynamic updates between controllers implementing these strategies enables sender and receiver to adapt their behaviour depending on the network failure rate.

A problem arises when updating the system while a message is in transit. In this situation the message was sent using the old piggybacking strategy, but the receiver, having been updated, is expecting a message using the new strategy. These problems are exhibited by controllers built using $T_\top$ and $T_\emptyset$. However, the manually built solution from [12] can be synthesised with $T = T_\emptyset \wedge ((startNewSpec \vee stopOldSpec) \implies \neg SendingMessage)$. The resulting controller provides the same guarantees as the one described in [12], except that in this case, the update strategy is constructed automatically.

6.5 Surveillance

Consider the case of a UAV required to search for suspicious activity in a critical area. It is required to fly in the 40 to 50 meter altitude range and transmit a picture for every potential target while avoiding obstacles. On low battery, it must return to any available base for recharging. During its mission, it is decided that the UAV must now immediately follow the first sighting that corresponds to a specific target rather than continuing its search. Furthermore, it must now regularly transmit pictures of suspicious activity. The new mission requires the UAV to fly in the 20 to 30 meter range and use an improved image recognition algorithm. In addition, because it is flying low, it must manage short-term weather forecasts to anticipate adverse conditions. New software modules must be loaded onto the UAV (the image processing module and a client for a weather forecast service) and the controller updated to achieve its new goals. Note that the non-overlapping height ranges imply that if the update is to occur in-flight, then a non-empty *transition period* will be required between the two missions (the descent from 40 to 30 meters) in which neither specification holds.

We produced different controllers by proposing variations of the transition properties. We first constructed transition requirements to replicate the condition in [13]. The result is an update controller that forces the UAV to fly back to the base where the mission started (while continuing to photograph and transmit potential targets) and only then to change mission and start out again. Transition requirement $T_\emptyset$ is a slightly weaker requirement that forces the change of mission when at any base (not necessarily the initial one) as this is the only state in which the flying requirements of both old and new missions are satisfied. We also produced even weaker transition requirements that allow in-flight updates. First we required that while neither specifications are being enforced, the UAV not change its x and y coordinates to avoid missing any suspicious activity while changing altitude: $(OldSpecStopped \wedge \neg NewSpecStarted) \implies \neg move$. We also included an additional requirement ensuring that when changing specification there should be no pictures of suspicious targets pending: $stopOldSpec \implies \neg PendingTransmissions$.

6.6 MetaSocket

The MetaSocket [11] is a socket that can be adapted through the insertion and removal of filters. In [11] four combinations of filters configurations are considered: DES64, DES128, DES64COM and DES128COM. As in [11] we generated a controller which is initially running in the DES64 system and then produced updates that are add or remove filters, one at a time. We modelled the different update scenarios and performed various chained updates, adding and removing filters to see if the technique produces bloated controllers of continuously increasing size. As described in section 5.2, the resulting controllers and control problems did not increase in size as further updates were performed.

6.7 Workflow Management - Simple Example

Workflow management systems automate and coordinate business processes to reduce costs and flow times. The problem of changing workflows at runtime to respond to changes in business goals is critical in this area [18, 31]. In [31] the essence of such a change is illustrated by an example in which a workflow that allows concurrent execution of billing and shipping tasks needs to be replaced by their sequential execution.

If a workflow update is requested halfway through the concurrent workflow and the ordering of tasks has occurred inconsistently with the new sequential workflow, the update may need to be deferred; this can be achieved automatically by requiring an empty transition ($T_{\emptyset}$). However, it is also possible to specify that the old specification should be dropped as soon as possible and that any items that cannot be processed according to the new specification be rolled-back as follows: $(\mathit{OldSpecStopped} \wedge \neg \mathit{NewSpecStarted}) \Rightarrow (G' \vee \mathit{rollback})$.

6.8 Workflow Management - Industrial Case Study

This case study is a business workflow provided by an industrial partner. The workflow manages a complex approval process that can take weeks to complete and is required to coordinate different areas of the company that review particular documents, create new ones and forward them to other actors.

The process can be divided into three main phases that correspond to the development and approval of three different documents: a Terms of Reference document (TOR), a Decision Support Document (DSD) and a Gate Form (GF). The documents themselves can be completed in any order, but revision and approval of the documents must be sequential: First TOR, then DSD and finally GF. At different points, actors responsible for the validation of documents can reject approvals or changes performed by other actors, making the worflow go back to a previous state.

The workflow was provided as an activity diagram and we converted it into a control problem. The environment model was designed to describe the various actors involved in the process with the assumption that each one, when required to complete, validate, or approve a document, would eventually do so. The goals encoded the precedence relations between different tasks and also the various decisions that can force tasks to be redone. We validated the description by ensuring that the controller that was synthesised was equivalent to the one provided by our industrial partner.

A postulated workflow change, that was validated with our industrial partner, involved inverting the order between DSD and TOR approvals and adding additional backward links depending on the Gatekeeper decisions upon reviewing documents. The change scenario required additional decisions regarding what to do when an instance is halfway through its approval process, in particular when TOR has been approved but not DSD. In this case, how should the approval process continue? Should the TOR be approved and then the DSD required to be approved again? Or should the DSD approval be cancelled to then continue with TOR and then DSD?

We constructed update controllers with different transition requirements. Naturally, transition requirement $T_{\top}$ allows an arbitrary treatment of half-processed instances. To produce update scenarios where pending proposals are forced to finish the flow satisfying the old specification, $T_{\emptyset}$ can be used. Finally, the requirement transition $T = (\mathit{OldSpecStopped} \wedge \neg \mathit{NewSpecStarted}) \implies (G' \vee \mathit{reviewForms})$, produce controllers for immediatelly satisfing new requirements, but if it is not possible, the system will ask to the leader proposal to resend every document.

6.9 Summary of Experience

Overall, 24 DCU synthesis problems were defined and solved, corresponding to different choices of transition requirements for each case study. In addition, we ran the Metasocket example only for testing a multiple updates scenario. We showed that resulting controllers after each update did not increase the amount of states avoiding a bloated controller. Table 1 summarises the case studies, the size of the environment model that describes them, the size of the resulting safe and live controller (C_u) and the time it took to compute it. The table also shows the size and synthesis time for the safe but not live controller (C_u^S) for each case study.

The majority of the safe and live controllers were synthesised in a few seconds. Noteworthy examples of computationally more complex controllers were those that had over 100 E_u states. Maximum synthesis time was close to 5 minutes.

Note that the controller synthesised when liveness was not required do not guarantee that update will eventually occur. In other words, in all case studies, to guarantee that dynamic update eventually occurs, an update controller that actively guides the system towards a safe transition state is needed. The cost of computing live controllers, when compared to safe but not live ones, varies significantly. In the majority of cases liveness did not greatly exacerbate the time for computing a safe controller, however the worst case was the workflow case study in which the synthesis time for a live controller was 20 times that of the safe one, approximately.

In addition, note that the synthesis algorithm is eager when attempting to achieve liveness. In other words, the controllers will attempt to perform an update in the least number of actions. This is very different from the controllers that only have safety requirements. The latter are synthesised to be maximal (allow every safe trace). This explains why the size of the live controllers were on average 60% smaller than the safe ones. A noteworthy example is the GSM case study in which for one of the transition requirements there was no reduction but a few more states.

7 Discussion and Related Work

Dynamic software update has been studied extensively and there are a plethora of different problems that must be addressed depending on the application domain, technology stack and the objective of the update (see [35] for a survey). Approaches to dynamic update typically assume that there is no specification change or it is not explicitly provided [36, 37] and hence that the behaviour is to be preserved (minus bugs to be fixed) (e.g., [38]), or that the specification is generic (e.g., [1, 39, 40, 41, 42, 43]) and not user provided. Examples of the latter, apart from ensuring the update does not lead to crashes, ensure safety (e.g., [44]) and data isolation between versions [45]. Quiescence [1] and related notions (e.g., [42, 43, 46]) do not originally deal with an explicit representation of the properties to be preserved,

Case Study	Transition Requirement	$\|E\| + \|E'\|$	$\|d\| + \|d'\|$	$\|G\| + \|G'\|$	$\|T\|$	$\|R\|$	$\|E_u\|$	$\|E_u\\|G_u\|$	$\|C\|$	$\|C_u^S\|$	C_u^S ms	$\|C_u\|$	C_u ms
Power Plant	$T_\top$	8	13	10	0	4	16	80	8	88	141	97	294
	$T_\emptyset$				8			66		72	153	52	241
	Leaves pump on				11			49		56	146	34	319
Railcab	$T_\top$	34	18	42	0	21	63	728	21	731	482	346	730
	$T_\emptyset$				18			663		668	570	276	466246
	brakes on update if late				5			750		744	289	360	763
Production Cell	$T_\top$	14	21	43	0	7	23	469	9	467	542	189	551
	No polished on update				3			419		411	270	174	617
	Remove polished or G'				5			798		795	440	227	5961
	Update while Empty				2			291		293	217	-	787
	$T_\emptyset$				10			163		164	263	103	541
	Keeps Old Spec				7			397		396	252	180	459
GSM	$T_\top$	17	2	2	0	8	25	81	8	76	99	83	117
	$T_\emptyset$				13			83		76	866	75	896
	No send while update				6			78		74	123	74	143
Workflow Management (simple)	$T_\top$	128	27	57	0	64	150	2782	22	2790	6412	592	7817
	$T_\emptyset$				17			176		182	3464	132	111895
Surveillance	$T_\top$	60	26	34	0	30	117	3226	56	3231	518	1287	992
	$T_\emptyset$				16			611		615	29960	365	30057
	No move while update				7			2900		2907	696	1236	1005
	No picture while update				7			1714		1718	684	1047	863
Workflow Management	$T_\top$	240	13	29	0	120	304	3425	64	3420	388	3196	816
	$T_\emptyset$				11			1272		1264	543	1567	687
	Resubmit documents				7			2881		2872	497	1844	6808

TABLE 1: Dynamic update controller synthesis report. $|d|$ and $|d'|$ are the number of fluents in the old/new specifications. $|G|$ and $|G'|$ are the number of logic operators in the old/new specifications. $|T|$ is the number of logic operators of the transition requirement. $|E_u\|G_u|$ is the number of states of the composition of E_u and the automata representation of G_u. The hyphen (-) indicates an unrealizable controller.

but have been used in conjunction with techniques that ensure semantic consistency (e.g., [47]).

The need for user-specified update properties has been recognised by [11] where property specification is discussed, [12, 48, 49] take manual or semi-automatic construction and then verify approaches, while [13, 14, 50, 51] discuss fully automatic synthesis.

A characterisation of different update property patterns was presented in [11] where it is argue that different patterns are applicable to different domains and update scenarios. For instance, in an emergency scenario, an update may be allowed to occur at any point in time and be required to occur as soon as possible, this corresponds to the "one-point" update property. In a planned update of, for instance, an automated production line, it may be required that the obligations of the current specification be satisfied before switching to the new specification. This scenario is referred to as "guided adaptation" as the system satisfying the current specification must be guided to fulfil its remaining obligations without acquiring new ones. The paper, however, does not discuss synthesis, i.e., the construction of strategies that fulfil these updates. In our work we can not only automatically produce strategies for the various update property patterns of [11] but also allow for additional scenarios they did not consider: a period in which neither the current specification nor the new one hold, but instead a transitional period in which an alternative temporary invariant must hold.

Others (e.g., [36, 48, 52, 53]) have considered the need for a transition period between specifications in which neither specification may hold. However, we provide a framework for formally specifying the requirements for the transition period and a synthesis algorithm that guarantees its preservation. For instance, Neamtiu et al. in [36] an update pattern for C programs is considered in which first the user sends a signal to the running program; after that, when the running system reaches a safe update point the initialization code begins; and finally, the initialization code "glues" the patch into the running program. The transition requirements here are implicit and the initialization code must be provided manually. Makris and Ryu [53] also have a phasing update structure similar to that of Nematiu et al. and ours. However, as before, the automation of these phases and guaranteed correctness of the construction is a key difference.

Zhang and Cheng [12] study the problem of building control update strategies. Their approach is semi-automatic, necessitating manual construction of "adaptation models" that can then be verified against requirements and used to construct programs. Ramirez et al. [49] build on this semi-automatic approach producing a tool capable of selecting and applying the best adaptation safe path that balances non-functional requirements, based on cost values. Our work could be extended to consider non-functional requirements by using quantitative control synthesis techniques such as [54] and [55].

As in [13, 14, 50, 51] we consider the use of synthesis to update a controller in a reactive system. In contrast to [13, 14] that have a fixed notion of correctness, we support user specified criteria for transition requirements. Furthermore, we propose a technique for dynamic update that *assures* that the system will reach a safe state by automatically computing the necessary strategy to take it to such a state even when the environment is not cooperative, while in [13, 14, 15] it is *assumed* that safe states will be reached. In [51], in addition to these liveness assumptions, there are also strong restriction on the kinds of update patterns that will occur. First, the assumptions for the new operating environment are required to be strictly stronger than those of the old environment. Second, they do not allow different transition semantics presented in [11], they only support "one-point" transitions. The notion of assuring liveness (the update eventually happens) is a key distinguishing feature of our approach with respect to other work on synthesis of update strategies. Note that in [11] liveness is recognized as a relevant aspect of update, but to the best of our knowledge this is the first technique that addresses this issue.

Perhaps, the closest work to that of this paper is [13] and [14]; for this reason we have compared our approach to theirs in Section 6 via case studies. [13] and [14] adopt a very natural and general correctness criteria which relieves the engineer from specifying transition requirements (in contrast to our approach) but at the cost of limiting the kind of updates that can be supported. In [13], if the system cannot return to its initial state and has not exhibited behaviour compatible with the new specification since the last initial state, then it is not possible to update. The update correctness criteria in [13] can be expressed as a transition requirement in this approach, but additionally, progress towards update can be guaranteed in our approach. In [14], weakened update criteria with respect to that of [13] are introduced to allow updates in systems where the initial state is not re-visited. However, there is no guarantee that the original correctness criteria (being equivalent to an offline update) holds. The lack of guarantees requires an engineer to validate the resulting controller. In our work, we involve the engineer upfront and support the provision of a specification of the correctness criteria for the update (T) which is then guaranteed by construction (e.g., see Power Plant in Section 6).

The problem of dynamic controller update is referred to by the discrete event systems (DES) community as *reconfiguration* (e.g., [2, 3, 4]). This community recognises the need for explicit requirements describing the transition from one configuration to the next. However, much of this work in this area assumes that the updates to be executed are known at design time, hence one controller is built in which all the updates to new controllers are precomputed. For example, in [3], the current DES has a special event for each available new configuration that when triggered starts the update. Both [2, 3] pre-compute reconfigurations that guarantee safety properties that can be expressed in our framework as part of T (see Definition 4.1)

In contrast, in this paper we do not assume that the new specification for which an update is to be performed is known at design time. One of the difficulties of dropping this assumption is that a strategy that must be viable for any current state of the running system must be synthesised, and an appropriate hot-swap mechanism must be put in place. These are novel aspects of the work presented in this paper.

A notable exception to assuming design time knowledge of updates in the DES community is [4] which is developed in a Petri net framework. However, they assume that while they are computing the update controller the current system is frozen. This is clearly unrealistic in certain scenarios and is a restriction that we overcome by hot swapping to a controller that in its first phase (see Figure 5) is capable of emulating the current controller and has a strategy for updating from any of its states.

The dynamic update of controllers is related to dynamic reconfiguration of software architectures. In controller update, one component (i.e., the controller) is replaced and the focus is on the behaviour that the system has as a result of the coordination that the controller provides. One of the aspects that may be coordinated is that of architectural reconfiguration. In our work, we abstract this issue to a $reconfigure$ command. However, this can be a complex process that may require its own planning and enactment. Dynamic reconfiguration of software architectures is an active field of study (e.g., [23, 56, 57, 58, 59]) that complements that of controller update.

Synthesis has been used extensively to guarantee code that is correct by construction (e.g., [60]). The fully automated nature of synthesis naturally leads to its potential application, not only at design-time, but also at runtime as a means to evolve software systems. Such evolution is not limited exclusively to adaptive systems. For instance in [61] the problem of evolving component assemblies is addressed by synthesising glue code (i.e. controllers). Although synthesis is performed without stopping the system, the new controller can only be put in place once the system has become quiescent.

Synthesis requires some sort of specification from which, through different reasoning techniques, to produce a solution. The result of synthesis is correct only to the extent that the specification is valid. Thus, synthesis techniques are, in principle, not resilient to errors in specifications or environments that evolve and diverge from the specification. The work described in this paper is also susceptible to invalid specifications. In the domain of adaptive systems, approaches that can detect and deal with such situations have been studied (e.g., [62, 63]) including how to learn new specifications at runtime (e.g., [64]). The approach described herein can be combined with such techniques.

In many situations, an unannounced change in the environment can occur and updating the controller to accommodate this change is desirable. For example, communication between a UAV and its base may be lost and as a result part of the interface that the UAV controller relies upon may be disabled. In these cases, a controller update must be realised immediately and it may be impossible to continue to guarantee the current goals or new goals. In [62] we present an approach for gracefully degrading the guarantees provided by the controller in such cases. However, the technique requires that the controller and specification of the degraded level preserve a refinement relation with the current controller and specification. Such a requirement can be restrictive and is not needed in our work. Furthermore, in [62] all degradation layers must be known, specified and synthesised at design time. Here, at runtime, a new unanticipated degradation step may be decided, specified (without requiring a refinement relation between E' and E), synthesised and deployed.

The linear time complexity of the DCU control problem when applied to deterministic environments provides an analytical argument to scalability. However, experimental validation remains to be done, and in particular to assess the practical need of introducing non-determinism as this can produce an exponential explosion.

Our work on dynamic controller update has the potential to address various of the problems identified in [65, 66, 67], working at a higher level of abstraction complementing approaches that achieve adaptation using continuous variable control techniques such as [68, 69, 70, 71].

In this paper we have restricted our discussion to avoid general liveness goals as part of the current and new specifications. This simplifies the presentation and also supports a linear resolution complexity for DCU control synthesis, if the control environment is deterministic. However, it is possible to allow further expressiveness in G, G' and T

without incurring the full penalty of solving control problems (2EXPTIME-COMPLETE). It is possible, for instance to reformulate Definition 4.1 to allow specifications G, G' and T to include subformulas of the form $\Box\Diamond\varphi$. Such a büchi acceptance criteria extends significantly expressiveness while remaining in a polynomial time complexity.

As mentioned, this paper is an extension of [16]. Here, we completely recast the dynamic update problem in terms of LTS and FLTL. In [16], we defined it over labelled transition Kripke structures. The reason for this change is that [16] required the new specification to subsume (in terms of the universe of state propositions) that of the old specification. This in turn means that as a system gets updated, the specifications get bloated. This restriction is no longer needed in the current formulation. In addition, we now provide a proof showing the completeness and the correctness of the presented technique. Finally, this paper also discusses five more case studies (Section 6), including a real workflow system provided by an industrial partner (see Section 6.8).

8 Conclusions

In this paper we define the correctness criteria for dynamic controller update. We present a solution to the problem of dynamically updating a controller to satisfy these criteria based on controller synthesis. The solution guarantees satisfaction of the new specification and any given transition requirements provided by the user. Moreover, by taking control of the system under the old specification and guiding it to a safe state in which the update can start, it ensures that the update will eventually occur satisfying the new specification.

As future work we plan to investigate how to increase the expressiveness of goals to include liveness but without having to pay the price of general synthesis. We also intend to investigate integration with other approaches that provide high-level adaptation capabilities to complex software systems, such as techniques for runtime learning of environment behaviour and adaptation for quantitative properties.

References

[1] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Trans. on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, Nov. 1990.

[2] H. E. Garcia and A. Ray, "State-space supervisory control of reconfigurable discrete event systems," *Int. Journal of Control*, vol. 63, no. 4, pp. 767–797, 1996.

[3] A. Nooruldeen and K. W. Schmidt, "State attraction under language specification for the reconfiguration of discrete event systems," *IEEE Trans. on Automatic Control*, vol. 60, no. 6, pp. 1630–1634, June 2015.

[4] R. Sampath, H. Darabi, U. Buy, and J. Liu, "Control reconfiguration of discrete event systems with dynamic control specifications," *IEEE Trans. on Automation Science and Engineering*, vol. 5, no. 1, pp. 84–100, 2008.

[5] *ICSE Symp. on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*. ACM/IEEE, 2006-2013.

[6] W. van der Aalst and K. van Hee, *Workflow Management: Models, Methods, and Systems*. Cambridge, MA, USA: MIT Press, 2004.

[7] H. Kress-Gazit and G. J. Pappas, "Automatically synthesizing a planning and control subsystem for the darpa urban challenge," in *2008 IEEE Int. Conf. on Automation Science and Engineering*, Aug 2008, pp. 766–771.

[8] P. J. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proc. of the IEEE*, vol. 77, no. 1, pp. 81–98, 1989.

[9] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive (1) designs," *Lecture notes in computer science*, vol. 3855, pp. 364–380, 2006.

[10] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "Synthesising non-anomalous event-based controllers for liveness goals," *ACM Tran. Softw. Eng. Methodol.*, vol. 22, 2013.

[11] J. Zhang and B. H. C. Cheng, "Specifying adaptation semantics," in *Proc. of the 2005 Workshop on Architecting Dependable Systems*, ser. WADS '05. New York, NY, USA: ACM, 2005, pp. 1–7.

[12] J. Zhang and B. H. Cheng, "Model-based development of dynamically adaptive software," in *Proc. of the 28th Int. Conf. on Software engineering*. ACM, 2006, pp. 371–380.

[13] C. Ghezzi, J. Greenyer, and V. P. La Manna, "Synthesizing dynamically updating controllers from changes in scenario-based specifications," in *Proc. of the 7th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2012, pp. 145–154.

[14] V. Panzica La Manna, J. Greenyer, C. Ghezzi, and C. Brenner, "Formalizing correctness criteria of dynamic updates derived from specification changes," in *Proc. of the 8th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 2013, pp. 63–72.

[15] I. Neamtiu and M. Hicks, "Safe and timely updates to multi-threaded programs," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 13–24.

[16] L. Nahabedian, V. Braberman, N. D'Ippolito, S. Honiden, J. Kramer, K. Tei, and S. Uchitel, "Assured and correct dynamic update of controllers," in *Proc. of the 11th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2016, pp. 96–107.

[17] C. Lewerentz and T. Lindner, Eds., *Formal Development of Reactive Systems - Case Study Production Cell*. London, UK, UK: Springer-Verlag, 1995.

[18] W. M. V.D Aalst and J. Stefan, "Dealing with workflow change: identification of issues and solutions," *Computer systems science and engineering*, vol. 15, no. 5, pp. 267–276, 2000.

[19] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, Jan. 2003.

[20] A. van Lamsweerde and E. Letier, "Handling obstacles in goal-oriented requirements engineering," *IEEE Trans. Softw. Eng.*, vol. 26, no. 10, pp. 978–1005, Oct. 2000.

[21] D. Giannakopoulou and J. Magee, "Fluent model checking for event-based systems," in *Proc. of the 11th ACM SIGSOFT Int. Symp. on Foundations of software engineering*, ser. ESEC/FSE-11, New York, NY, USA, 2003, pp. 257–266.

[22] L. de Alfaro and T. A. Henzinger, "Interface automata," in *ESEC / SIGSOFT FSE*. ACM, 2001, pp. 109–120.

[23] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic, "Plasma: a plan-based layered architecture for software model-driven adaptation," in *Proc. of the IEEE/ACM Int. Conf. on Automated software engineering*, 2010, pp. 467–476.

[24] F. Alvares, E. Rutten, and L. Seinturier, "A domain-specific language for the control of self-adaptive component-based architecture," *Journal of Systems and Software*, 2017.

[25] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proc. of the 16th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, 1989, pp. 179–190.

[26] D. Ciolek, V. A. Braberman, N. D'Ippolito, N. Piterman, and S. Uchitel, "Interaction models and automated control under partial observable environments," *IEEE Trans. Software Eng.*, vol. 43, no. 1, pp. 19–33, 2017.

[27] N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel, "Mtsa: The modal transition system analyser," in *Proc. of the 23rd IEEE/ACM Int. Conf. on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 475–476.

[28] "Mtsa synthesis tool and case studies," 2017. [Online]. Available: http://mtsa.dc.uba.ar/updateControllers/2017-tse.html

[29] J. Magee and J. Kramer, *Concurrency: state models & Java programs*. Wiley New York, 2006.

[30] R. M. Keller, "Formal verification of parallel programs," *Communications of the ACM*, vol. 19, pp. 371–384, July 1976.

[31] C. Ellis, K. Keddara, and G. Rozenberg, "Dynamic change within workflow systems," in *Proc. of Conf. on Organizational Computing Systems*, ser. COCS '95, 1995, pp. 10–21.

[32] Microsoft. (2017, Mar.) Windows workflow foundations. dynamic update. Https://goo.gl/KT6gtJ.

[33] M. Kontitsis, K. P. Valavanis, and N. Tsourveloudis, "A uav vision system for airborne surveillance," in *Proc. of the IEEE Int. Conf on*

Robotics and Automation, 2004. ICRA'04., vol. 1, 2004, pp. 77–83.

[34] Paderborn, "New rail technology paderborn," Aug. 2014, http://www.railcab.de/.

[35] H. Seifzadeh, H. Abolhassani, and M. S. Moshkenani, "A survey of dynamic software updating," *Journal of Software: Evolution and Process*, vol. 25, no. 5, pp. 535–568, 2013.

[36] I. Neamtiu, M. W. Hicks, G. P. Stoyle, and M. Oriol, "Practical dynamic software updating for C," in *Proc. of the Conference on Programming Language Design and Implementation*, 2006, pp. 72–83.

[37] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A powerful live updating system," in *Proc of the 29th Int. Conf. on Software Engineering*. IEEE Computer Society, 2007, pp. 271–281.

[38] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," in *Proc. of the Int. Conf. on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 612–621.

[39] J. Shen, X. Sun, G. Huang, W. Jiao, Y. Sun, and H. Mei, "Towards a unified formal model for supporting mechanisms of dynamic component update," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 80–89, Sep. 2005.

[40] H. Chen, J. Yu, C. Hang, B. Zang, and P.-C. Yew, "Dynamic software updating using a relaxed consistency model," *IEEE Trans. on Soft. Eng.*, vol. 37, no. 5, pp. 679–694, Sept 2011.

[41] A. Orso, A. Rao, and M. J. Harrold, "A technique for dynamic updating of java software," in *Proc of the Int. Conf. on Software Maintenance, 2002.* IEEE, 2002, pp. 649–658.

[42] A. Anderson and J. Rathke, "Migrating protocols in multi-threaded message-passing systems," in *Proc. of the 2Nd Int. Workshop on Hot Topics in Software Upgrades*, ser. HotSWUp '09. New York, NY, USA: ACM, 2009, pp. 8:1–8:5.

[43] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Trans. on Software engineering*, vol. 22, no. 2, 1996.

[44] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: A vm-centric approach," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 1–12.

[45] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "Mutatis mutandis: Safe and predictable dynamic software updating," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 4, aug 2007.

[46] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates," *IEEE Trans. on Software Engineering*, vol. 33, no. 12, pp. 856–868, 2007.

[47] F. Banno, D. Marletta, G. Pappalardo, and E. Tramontana, "Handling consistent dynamic updates on distributed systems," in *Proc. of the IEEE Symp. on Computers and Communications (ISCC),*. IEEE, 2010, pp. 471–476.

[48] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, "Specifying and verifying the correctness of dynamic software updates," in *Proc. of the 4th Int. Conf. on Verified Software: Theories, Tools, Experiments*, ser. VSTTE'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 278–293.

[49] A. J. Ramirez, B. H. Cheng, P. K. McKinley, and B. E. Beckmann, "Automatically generating adaptive logic to balance non-functional tradeoffs during reconfiguration," in *Proc. of the 7th Int. Conf. on Autonomic Computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 225–234.

[50] L. Baresi and C. Ghezzi, "The disappearing boundary between development-time and run-time," in *Proc. of the FSE/SDP Workshop on Future of Software Engineering Research*. New York, NY, USA: ACM, 2010, pp. 17–22.

[51] S. An, X. Ma, C. Cao, P. Yu, and C. Xu, "An event-based formal framework for dynamic software update," in *IEEE Int. Conf. on Software Quality, Reliability and Security*. IEEE, 2015, pp. 173–182.

[52] D. Gupta, "On-line software version change," Ph.D. dissertation, 1994.

[53] K. Makris and K. D. Ryu, "Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 327–340, 2007.

[54] R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann, "Better quality in synthesis through quantitative objectives," in *Proc. of the 21st Int. Conf. on Computer on Aided Verification, CAV 2009, Grenoble, France.*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 140–156.

[55] D. Sykes, W. Heaven, J. Magee, and J. Kramer, "Exploiting non-functional preferences in architectural adaptation for self-managed systems," in *Proc. of the 2010 ACM Symp. on Applied Computing*. ACM, 2010, pp. 431–438.

[56] N. Arshad, D. Heimbigner, and A. L. Wolf, "Deployment and dynamic reconfiguration planning for distributed software systems," *Software Quality Journal*, vol. 15, no. 3, pp. 265–281, 2007.

[57] D. Sykes, W. Heaven, J. Magee, and J. Kramer, "From Goals to Components: A Combined Approach to Self-Management," in *Proc. of the Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2008.

[58] L. Baresi, C. Ghezzi, X. Ma, and V. P. La Manna, "Efficient dynamic updates of distributed components through version consistency," *IEEE Trans. on Soft. Eng.*, 2016.

[59] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel, "Morph: A reference architecture for configuration and behaviour self-adaptation," in *Proc. of the 1st Int. Workshop on Control Theory for Software Engineering*, ser. CTSE 2015, 2015, pp. 9–16.

[60] J. Greenyer, C. Brenner, M. Cordy, P. Heymans, and E. Gressi, "Incrementally synthesizing controllers from scenario-based product line specifications," in *Proc. of the 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, New York, NY, USA, 2013, pp. 433–443.

[61] P. Pelliccione, M. Tivoli, A. Bucchiarone, and A. Polini, "An architectural approach to the correct and automatic assembly of evolving component-based systems," *J. Syst. Softw.*, vol. 81, no. 12, pp. 2237–2251, Dec. 2008.

[62] N. D'Ippolito, V. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel, "Hope for the best, prepare for the worst: Multi-tier control for adaptive systems," in *Proc. of the 36th Int. Conf. on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 688–699.

[63] P. Vromant, D. Weyns, S. Malek, and J. Andersson, "On interacting control loops in self-adaptive systems," in *Proc. of the 6th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '11. New York, NY, USA: ACM, 2011, pp. 202–207.

[64] D. Sykes, D. Corapi, J. Magee, J. Kramer, A. Russo, and K. Inoue, "Learning revised models for planning in adaptive systems," in *Proc. of the Int. Conf. on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 63–71.

[65] B. H. Cheng, R. De Lemos, H. Giese, P. Inverardi, J. Magee *et al.*, "Software engineering for self-adaptive systems: A research roadmap," in *Software engineering for self-adaptive systems*. Springer, 2009, pp. 1–26.

[66] A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein *et al.*, "Software engineering meets control theory," in *Proc. of the 10th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 2015, pp. 71–82.

[67] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio, "Control-theoretical software adaptation: A systematic literature review," *IEEE Trans. on Software Engineering*, 2017.

[68] A. Bergen, N. Taherimakhsousi, and H. A. Müller, "Adaptive management of energy consumption using adaptive runtime models," in *Proc of the 10th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2015, pp. 120–126.

[69] A. Filieri, H. Hoffmann, and M. Maggio, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *Proc of the 36th Int. Conf. on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 299–310.

[70] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio, "Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements," in *Proc. of the 26th IEEE/ACM Int. Conf. on Automated Software Engineering, 2011*. IEEE Computer Society, 2011, pp. 283–292.

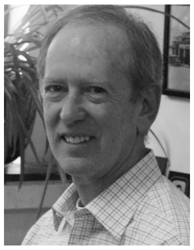
[71] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva, "Power optimization in embedded systems via feedback control of resource allocation," *IEEE Trans. on Control Systems Technology*, vol. 21, no. 1, pp. 239–246, Jan 2013.

Leandro Nahabedian is a teaching assitant at Universidad de Buenos Aires. He received the undergraduate computer science degree from Universidad de Buenos Aires. He is currently working toward the PhD degree in computer science at Universidad de Buenos Aires. His research areas interests are in the broad area of software engineering, including formal methods for modeling and designing self-adaptive software. Leandro has been distinguished with the Best Paper Award at SEAMS 2016.

Victor Braberman is an associate professor at Universidad de Buenos Aires, Facultad de Ciencias Exactas y Naturales, Departamento de Computación and he is a CONICET researcher working at Instituto de Investigación en Ciencias de la Computación (ICC), Buenos Aires, Argentina. Dr. Braberman received PhD in Computer Science at Universidad de Buenos Aires. His research interests include innovative uses of formal abstractions for the analysis, construction and understanding of software intensive systems. His publication track record includes publications at the top Software Engineering conferences and journals. He serves regularly as PC/PB member for flagship Software Engineering and Formal Methods conferences. He is a current member of IEEE's Transactions on Software Engineering's Editorial Board. He has also significant industrial experience in consultancy and in leading R&D projects for software companies.

Nicolás D'Ippolito is a Professor at Universidad de Buenos Aires and a CONICET researcher. He received his undergraduate Computer Science degree from University of Buenos Aires and his PhD in Computing from Imperial College London. His research interests fall in multiple areas Control Theory, Software Engineering, Adaptive Systems, Model Checking and Robotics. Specifically, he is interested in behaviour modelling, analysis and synthesis applied to requirements engineering, adaptive and autonomous systems, software architectures design, validation and verification. Dr. D'Ippolito regularly publishes in top conferences and journals in many areas. He has served as PC member for flagship conferences a number of times and has organised many events in top venues.

Shinichi Honiden received the PhD degree from Waseda University in 1986. He was at Toshiba Corporation from 1978 to 2000. Since 2000, he has joined the National Institute of Informatics (NII) as a professor. His research interests include software engineering, agent technology, and pervasive computing. He is the deputy director general at NII in Tokyo, Japan, a professor in the Graduate School of Information Science and Technology at the University of Tokyo, and Director of the Center for Global Research in Advanced Software Science and Engineering (GRACE Center). He is a steering committee member of the International Conference on Automated Software Engineering since 2006. In 2006, he received the ACM Recognition of Service Award from the Association for Computing Machinery. In 2012, he received The Commendation for Science and Technology from the Japanese Minister of Education, Culture, Sports, Science and Technology.



Jeff Kramer is a Professor at Imperial College London. He was Head of the Department of Computing from 1999 to 2004, Dean of the Faculty of Engineering from 2006 to 2009 and the Senior Dean from 2009 to 2012. His research work is primarily concerned with software engineering, focusing on software architecture, behaviour analysis, the use of models in requirements elaboration and architectural approaches to adaptive software systems. He was a principal investigator of research projects that developed the CONIC and DARWIN architectural environments for distributed programming and of associated research into software architectures and their analysis. Jeff was Program Co-chair of ICSE '99, Chair of the ICSE Steering Committee from 2000 to 2002, and General Co-chair of ICSE 2010 in Cape Town. He was Editor in Chief of IEEE TSE from 2006 to 2009, received the Most Influential Paper Award at ICSE 2003, and was awarded the 2005 ACM SIGSOFT Outstanding Research Award and the 2011 ACM SIGSOFT Distinguished Service Award. He is co-author of books on Concurrency and on Distributed Systems and Computer Networks, and the author of over 200 journal and conference publications. Jeff is a Fellow of the Royal Academy of Engineering, a Chartered Engineer, Fellow of the IET, Fellow of the ACM, Fellow of the BCS, Fellow of the City and Guilds of London Institute and a Member of Academia Europaea.

Kenji Tei is an associate professor at National Institute of Informatics. Dr. Kenji Tei received Ph.D. in Engineering from Waseda University, Japan in 2008. He joined Department of Information and Computer Science in Waseda University, and National Institute of Informatics as a research assistant. From 2008 to 2010, he was assistant professor at Media Network Center in Waseda University, and project assistant professor at National Institute of Informatics. From 2010 to 2015, he was an assistant professor at National Institute of Informatics. He is interested in software engineering for self-adaptive software, in particular, models@runtime techniques and software architecture for self-adaptive software.

Sebastián Uchitel is a Professor at University of Buenos Aires, a CONICET researcher and holds a Readership at Imperial College London. He received his undergraduate Computer Science degree from University of Buenos Aires and his Phd in Computing from Imperial College London. His research interests are in behaviour modelling, analysis and synthesis applied to requirements engineering, software architecture and design, validation and verification, and adaptive systems. Dr. Uchitel was associate editor of the Transactions on Software Engineering and is currently associate editor of the Requirements Engineering Journal and the Science of Computer Programming Journal. He was program co-chair of ASE'06 and ICSE'10, and General Chair of ICSE'17 helded in Buenos Aires. Dr Uchitel has been distinguished with the Philip Leverhulme Prize, an ERC StG Award, the Konex Foundation Prize and the Houssay Prize.