Assured and Correct Dynamic Update of Controllers

L. Nahabedian†, V. Braberman†, N. D’Ippolito†, S. Honiden†, J. Kramer†, K. Tei†, S. Uchitel†
† Department of Computing, Imperial College London, UK
‡ Departamento de Computación, FCEN, Universidad de Buenos Aires, Argentina
† National Institute of Informatics, Japan

ABSTRACT

In many application domains, continuous operation is a desirable attribute for software-intensive systems. As the environment or system requirements change, so the system should change and adapt without stopping or unduly disturbing its operation. There is, therefore, a need for sound engineering techniques that can cope with dynamic change.

In this paper we address the problem of dynamic update of controllers in reactive systems when the specification (environment assumptions, requirements and interface) of the current system changes. 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. Indeed, using controller synthesis we show how to automatically build a controller that guarantees both progress towards update and safe update. Seven case studies have been implemented to validate the approach.

Categories and Subject Descriptors

2.4 [Software Engineering]: Software Verification

Keywords

Controller Synthesis, Dynamic Update, Adaptive Systems

1. INTRODUCTION

Continuous operation is a requirement common to many application domains. There is therefore a need for sound engineering techniques that can change a system in the face of environment and requirements changes without stopping or disrupting its operation. This challenge has been addressed in complementary ways, starting with dynamic component update and reconfiguration [29] and more recently with adaptive systems design [43].

When is it safe to change a running system? One conservative answer to this question is “when components are not involved in any interactions”; this was formalised through notions of quiescence [29] and later tranquility [51]. For instance, in [29], components in a distributed system are designed to accept passivate messages which, by disallowing components to initiate new transactions, attempt to, but do not guarantee, progress towards quiescence. Many other techniques have been developed (such as [3, 24]), however, most do not include explicit consideration of specification change [8] and of whether or not the update is correct with respect to the specification.

For runtime specification change in reactive systems, the question is “when is it safe to hot swap the current controller with a new one such that the new specification is enforced?”. Zhang and Cheng argue that the update conditions “must be explicitly captured at the requirements level” [52]. They show that different domains and scenarios require distinct update conditions. Sometimes update may be allowed to occur cleanly at any point in time (usually as soon as possible), and in others obligations of the current specification must be honoured before switching to the new specification. Requirements to guide the system from one specification to another (as in [4]) or to ensure graceful update (as in [13]) may be appropriate in certain settings. These update strategies generally use verification to ensure correctness, with (manual) construction of “adaptation models”.

Automatic synthesis of controller update strategies for reactive systems has been proposed in [20, 30, 2] amongst others. However, in all cases it is assumed that the system being executed eventually reaches (what each technique considers) a safe state. This liveness assumption of eventually reach a safe state is very strong. Zhang and Cheng [52] recognise that a system may need to be guided to a safe updatable state; however it is the user that is required to produce the strategy to do so. Furthermore, this may not be trivial if the environment is not cooperative!

A question that has not been addressed previously in the problem of runtime specification change is “when is it safe to reconfigure the controller’s environment?”. A change in the specification means a change in system goals and assumptions. A change in the latter is typically driven by a change in the environment (e.g., the component or services that are available to the controller change). As with updating controllers, we believe that conditions for reconfiguring the
software architecture on which the controllers rely upon must be explicitly captured at the requirements level.

Indeed, the problem of dynamic controller update can be thought of as the problem of developing a third controller (an update controller) that manages the hot-swap of the current controller with the controller for the new specification while satisfying additional requirements on when this should happen and when the environment should be reconfigured. However, conceptually simpler, two of the three controllers can be thought of as one: the problem is to develop an update controller that can take control of the system at any point, continue to satisfy the old specification while managing the environment reconfiguration and move on to satisfy the new specification.

In this paper, taking the latter view, we i) show how requirements for describing the transition between specification changes can be described restricting when the change can occur and also when any reconfiguration needed should occur; ii) formalise the behaviour of a system in which a controller is hot-swapped and in which a controller can reconfigure it’s environment; iii) define a correctness criteria for dynamic controller update; and iv) define an automatic procedure, based on discrete event control theory (e.g. \cite{31, 38, 16}), for computing an update controller that handles hot-swapping, reconfiguration and the new specification.

The rest of the paper is structured as follows. We start with a motivating example (Section 2). Then, provide the formal definitions (Section 3) required to formalize the problem of dynamic controller update (Section 4). Later, our approach solution is discuss in Section 5. We continue with validation in Section 6 and conclude with a discussion and related work.

2. MOTIVATING EXAMPLE

Assume an industrial automation scenario \cite{31} in which a robotic arm moves raw products arriving on an In tray through a number of different tools to then drop 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 of the form $\langle E, G, A \rangle$. $A$ is the set of events the controller can execute, for instance pick/p, 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 paint product $x$ ($\text{paint}(x)$) will respond with $\text{paintOK}(x)$ or $\text{paintNOK}(x)$ representing success or failure in painting $x$. 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 has been stamped as faulty. A formalisation of some of the goals may be:

$\text{ToolOrder} \equiv \forall x \cdot (\text{Cleaned}(x) \Rightarrow \text{Polished}(x)) \land \text{Polished}(x) \Rightarrow \text{Drilled}(x))$

$\text{ToolsRequired} \equiv \forall x \cdot \text{out}(x) \Rightarrow (\text{Faulty}(x) \lor (\text{Drilled}(x) \land \text{Polished}(x) \land \text{Cleaned}(x) \land \neg \text{Stamped}(x)))$

$\text{NoProcessingIfFaulty} \equiv \forall x \cdot (\text{Faulty}(x) \Rightarrow \neg (\text{drill}(x) \lor \text{polish}(x) \lor \text{clean}(x)))$

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 is found to be faulty, 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 that 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 is in many cases unacceptable, for instance in factories down-time can have serious economic consequences.

Assume that for business reasons, a polishing tool is to be changed by a paint tool. Such a change entails a number of re-orderings in the production workflow. The new production workflow is captured by specification $\langle E', G', A' \rangle$ 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:

$\text{ToolOrder'} \equiv \forall x \cdot (\text{Drilled}(x) \Rightarrow \text{Painted}(x)) \land (\text{Painted}(x) \Rightarrow \text{Cleaned}(x))$

$\text{ToolsRequired'} \equiv \forall x \cdot \text{out}(x) \Rightarrow (\text{Faulty}(x) \lor (\text{Drilled}(x) \land \text{Painted}(x) \land \text{Cleaned}(x) \land \neg \text{Stamped}(x)))$

How should the current controller satisfying $\langle E, G, A \rangle$ be updated to now satisfy $\langle E', G', A' \rangle$? 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 \cite{50}, the answer 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 processed according to the new requirements $G'$? Should a polished but not clean product be painted and then placed on the Out tray? Should it be discarded without further processing? Or should the update be delayed until there are no painted 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 an event to signal when the old specification is dropped (stopOldSpec) and another to signal from when the new specification is to be guaranteed (startNewSpec). Furthermore, we assume propositions OldSpecStopped and NewSpecStarted that indicate if these events have occurred.

One possible transition requirement is that no polished should be on the line ($T_1 = \text{startNewSpec} \iff \forall x \cdot \text{OnProductionLine}(x) \Rightarrow \neg \text{Polished}(x)$). Another transition requirement could be that products are to be either output according to the new specification $G'$ or trashed. This allows, for instance, partially processed products that cannot be continued to be processed according to $G'$ be dealt with.

$T_2 \equiv (\text{OldSpecStopped} \land \neg \text{NewSpecStarted}) \Rightarrow (\text{ToolOrder'}')$

$\text{ToolsRequired''} \equiv \forall x \cdot \text{out}(x) \Rightarrow (\text{Faulty}(x) \lor \text{Stamped}(x) \lor (\text{Drilled}(x) \land \text{Painted}(x) \land \text{Cleaned}(x) \land \neg \text{Polished}(x)))$

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 \( \implies \forall x. \neg \text{BeingPolished}(x) \)). 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 \( G \) (for instance finishing off polished products according to \( G \)) while ensuring that eventually an update state is reached (for instance polished products on the line). Thus, the solution to how the system is updated from \((E, G, A)\) to \((E', G', A')\) 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, in this paper we present a fully automated technique that can guarantee a correct update of the controller for the production plant scenario. Informally, the input to the technique we present is the current specification \((E, G, A)\), the controller currently supervising the production plant \( (C') \), the new specification \((E', G', A')\), and transition requirements \((T)\). The output is an update controller \( (C')' \) that assures that the resulting system satisfies the following requirements:

1. \( C \) can be hot-swapped by \( C' \) at any point in time.
2. \( G \) will continue to hold until \( C' \) signals stopOldSpec.
3. \( T \) that prescribes stopOldSpec, startNewSpec and reconfigure holds.
4. \( G' \) will hold once \( C' \) signals startNewSpec.
5. Once \( C \) and \( C' \) are hot-swapped, the following will 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 is offline and 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 [28] 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 would compute 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 \( C' \) will execute 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. PRELIMINARIES

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

**Definition 3.1. (Labelled Transition Kripke Structures)** A Labelled Transition Kripke Structure (LTKS) \( E \) is a tuple \((S_E, A_E, \Delta_E, P_E, v_E, e_0)\), where \( S_E \) is a finite set of states, \( A_E \subseteq \text{Act} \) is its communicating alphabet, \( \text{Act} \) is the universe of all observable events, \( P_E \subseteq \text{Act} \) is a set of propositions, \( \text{Prop} \) is the universe of all propositions, \( \Delta_E \subseteq (S_E \times A_E \cup \{\tau\} \times S_E) \) is a transition relation and \( \tau \) an unobservable event, \( v : S_E \rightarrow 2^P_E \) is a valuation function for states, and \( e_0 \in S_E \) is the initial state. We denote \( \Delta_E(e) = \{ (\ell') | (e, \ell, e') \in \Delta_E \} \). A path of \( E \) is a sequence \( \pi = s_0, e_0, s_1, \ldots \) where for every \( i \geq 0 \) we have \((s_i, e_i, s_{i+1}) \in \Delta_E \). If a path \( \pi \) is finite, it ends with a state \( s_n \). An event-trace (resp. state-trace) \( w \) is the sequence that results from removing all events not in \( A_E \) (resp. \( S_E \)) from \( \pi \). For a state \( e \in S_E \), we denote changing the initial state of \( E \) from \( s_0 \) to \( e \) as \( E(e) \).

We say that \( E \) is deterministic if \( (e, \ell, e') \in \Delta \) and \( (e, \ell, e') \in \Delta \implies e' = e'' \), and is deadlock-free if for all \( e \in S \) there exists \((e, \ell, e') \in \Delta \).

**Definition 3.2. (Parallel Composition)** The parallel composition \( E|C \) of two LTKS \( E = (S_E, A_E, \Delta_E, P_E, v_E, e_0) \) and \( C = (S_C, A_C, \Delta_C, P_C, v_C, e_0) \) is an LTKS \((S_{||}, A_E \cup A_C, \Delta_{||}, P_E \cup P_C, v_{||}(e_0, e_0))\) such that \( S_{||} \) is a set of pairs
of states consistent based on propositions (i.e. $S_{\mathbb{E}} = \{(e, c) \in (S_{\mathbb{E}} \times S_{\mathbb{C}}) / \forall e \in E \land \mathbb{P}_C = v_{\mathbb{C}}(c) \cap \mathbb{P}_B \}$), $v_{\mathbb{E}} : (S_{\mathbb{E}} \times S_{\mathbb{C}}) \rightarrow 2^{E_{\mathbb{E}}}$ define as $v_{\mathbb{E}}((e, c)) = v_{\mathbb{E}}(e) \cup v_{\mathbb{E}}(c)$, and $\Delta_{\mathbb{E}}$ is the smallest relation that satisfies the rules below.

\[
\begin{align*}
(e, \ell, e') &\in \Delta_{\mathbb{E}} \quad ((e, c), \ell, (e', c)) \in \Delta_{\mathbb{E}} \quad (e \in A_{\mathbb{E}} \setminus A_{\mathbb{C}}) \\
(e, \ell, c') &\in \Delta_{\mathbb{C}} \quad ((e, c), \ell, (e', c')) \in \Delta_{\mathbb{C}} \quad (e \in A_{\mathbb{C}} \setminus A_{\mathbb{E}}) \\
(e, \ell, e') &\in \Delta_{\mathbb{E}} \quad ((e, c), \ell, (e', c')) \in \Delta_{\mathbb{C}} \quad (e \in A_{\mathbb{E}} \cap A_{\mathbb{C}})
\end{align*}
\]

The definition of legally-based interface automata [14] allows modeling controllability. $C$ is legal with respect to $E$ if it does not block $E$ from performing events that $C$ does not control.

**Definition 3.3. (Legal LTKS)** Let $C = (S_{\mathbb{C}}, A_{\mathbb{C}}, \Delta_{\mathbb{C}}, P_{\mathbb{C}}, v_{\mathbb{C}}, c_0)$ and $E = (S_{\mathbb{E}}, A_{\mathbb{E}}, \Delta_{\mathbb{E}}, P_{\mathbb{E}}, v_{\mathbb{E}}, e_0)$ be LTKSs, and $A \subseteq A_{\mathbb{E}}$ be a set of events that $C$ does not control. We say that $C$ is a legal LTKS for $E$ with respect to $A$ if for all $(e, c) \in E \cap C$, $\Delta_{\mathbb{C}}(e) \cap (A_{\mathbb{E}} \setminus A) \subseteq \Delta_{\mathbb{C}}(c) \cap (A_{\mathbb{E}} \setminus A)$ holds.

The interrupt operation defines the replacement of a model by another, and a relation between their states that identifies the initial state of the replacing model once it replaces the old one at execution time. More formally we have:

**Definition 3.4. (Interrupt operator)** Let $E = (S_{\mathbb{E}}, A_{\mathbb{E}}, \Delta_{\mathbb{E}}, P_{\mathbb{E}}, v_{\mathbb{E}}, e_0)$ and $N = (S_{\mathbb{N}}, A_{\mathbb{N}}, \Delta_{\mathbb{N}}, P_{\mathbb{N}}, v_{\mathbb{N}}, n_0)$ be LTKSs, and $R \subseteq (S_{\mathbb{E}} \times S_{\mathbb{N}})$ be a binary relation between states of $E$ and $N$. The interrupt operation $E \vec{R} N$ of $E$ with $N$ with $R$ and $\alpha$ is an LTKS defined as $(S_{\mathbb{E}} \cup S_{\mathbb{N}}, A_{\mathbb{E}} \cup A_{\mathbb{N}}, \Delta_{\mathbb{E}} \cup \Delta_{\mathbb{N}}, P_{\mathbb{E}} \cup P_{\mathbb{N}}, v_{\mathbb{E}} \cup v_{\mathbb{N}}, e_0)$, where $\Delta_{\mathbb{E}}$ is the smallest relation that satisfies the rules below:

\[
\begin{align*}
(e, \ell, c') &\in \Delta_{\mathbb{E}} \quad ((e, c), \ell, (e', c')) \in \Delta_{\mathbb{E}} \quad (e, n) \in R \\
(n, \ell, e') &\in \Delta_{\mathbb{N}} \quad ((n, \ell), (e', e')) \in \Delta_{\mathbb{N}} \quad (e, n) \in R
\end{align*}
\]

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

**Definition 3.5. (Relabelling operator)** Let $E = (S_{\mathbb{E}}, A_{\mathbb{E}}, \Delta_{\mathbb{E}}, P_{\mathbb{E}}, v_{\mathbb{E}}, e_0)$ be an LTKS and $f : A \subseteq \text{Act} \rightarrow A' \subseteq \text{Act}$ be a function. The relabelling operation $[E]_f$ of $E$ over $f$ is an LTKS $(S_{\mathbb{E}}, A', \Delta_f, P_{\mathbb{E}}, v_{\mathbb{E}}, e_0)$, where $\Delta_f$ is the smallest relation that satisfies the rule below:

\[
\begin{align*}
(e, \ell, e') &\in \Delta_{\mathbb{E}} \quad ((e, f(\ell)), (e', f(e'))) \in \Delta_{\mathbb{E}} \quad \ell \in \text{Dom}(f)
\end{align*}
\]

We fix Linear Temporal Logic (LTL) [39] for describing properties. Let $P$ be a set of propositions. An LTL formula is defined inductively using the standard Boolean connectives and temporal operators $X$ (next), $U$ (strong until) as follows: $\phi ::= p \mid \neg \phi \mid \phi \lor \psi \mid X\phi \mid \phi U \psi$ where $p \in P$. We define $\phi \land \psi$ as $\neg (\neg \phi \lor \neg \psi)$, $\phi$ (eventually) as $T \phi$, $\phi$ (always) as $\neg \diamond \neg \phi$, and $\phi W \psi$ (weak until) as $\phi U (\psi \lor \phi)$.

LTL formulae satisfaction is computed over state-traces of an LTKS, is standard [6] and omitted. In this paper we assume states in traces are extended to contain a proposition indicating the last event that occurred. This allows expressing properties over events and propositions, supporting more succinct formalisations.

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

**Definition 3.6 (LTKS Control [16]).** Let $E = (S_{\mathbb{E}}, A_{\mathbb{E}}, \Delta_{\mathbb{E}}, P_{\mathbb{E}}, v_{\mathbb{E}}, e_0)$ be an environment model in the form of an LTKS, $A \subseteq A_{\mathbb{E}}$ be a set of controllable actions, and $G$ be a controller goal in the form of an LTL property. A solution for the LTKS control problem with specification $(E, G, A)$ is an LTKS $C$ such that $C$ is a legal LTKS for $E$ with respect to $A$, $E[C] = \text{deadlock free}$, and $E[C] \models G$.

4. PROBLEM STATEMENT

In this section we formalise the problem of synthesising an update controller which can be dynamically hot swapped into a running system and guarantees that the system will transition in a correct manner from an old system specification and environment configuration to a new one. For this we must first formalise the concept of hot-swapping a controller, the notion of a controller commanding its environment reconfiguration and what is meant by correctness.

4.1 Hot-swapping Controllers

Consider a controller $C$ executing in an environment $E$. The behaviour of such a system is captured by the term $C \parallel E$. Assume that $C$ is to be hot-swapped by a new controller $C'$. The initial state of $C'$ may depend on the current state of $C$, hence we require as part of the hot-swap operation that a mapping from states of $C$ and $C'$ be given. We model the hot-swap of $C$ with $C'$ as $(C \parallel C')^{\text{hotSwap}}$ using the interrupt operator.

At the top of Figure 2 we depict a controller $C$ that senses ambient light (low; med; high) and switches a light bulb on and off depending on whether ambient light is low or not. At the bottom of the same figure, we show a controller $C'$ that switches the light bulb only if ambient light is either low or medium. The combination of both controllers plus the dotted hotSwap transitions is the result of $(C \parallel C')^{\text{hotSwap}}$ where the hotSwap transitions coincide with the mapping of states in $C$ to states in $C'$ by $R$.

On a methodological note, the term $C^{\text{hotSwap}}_R$ has as communicating alphabet the union of the alphabets of $C$ and $C'$. This means that all events in the communicating alphabet of $C$ and not in that of $C'$ (resp. in $C'$ and not in $C$) are restricted from occurring after (resp. before) hotSwap (see rules for parallel composition and interrupt in the previous section). For instance, in the motivating example, $C$ has in its alphabet paint($x$) but not polish($x$) which is in the alphabet of $C'$. The term $C^{\text{hotSwap}}_R$ will
prohibit the occurrence of paint(x) before hotSwap and the occurrence of polish(x) after hotSwap.

We will assume that hotSwap is not in the alphabet of C, C’ nor E, and that R covers all states in C. Consequently, in (C′ | hotSwap C′)E the event hotSwap is unconstrained. Indeed, this models that hotSwap will be triggered by the underlying infrastructure which will use R to set the current state of C′ according to that of C.

Finally, we require the controller to be an LTKS with no propositions. This is akin to saying that the controller cannot sense nor modify propositions directly, it can only sense via event synchronisation.

4.2 Controlling Reconfiguration

Dynamically changing controllers due to a change of specification must support controlled dynamic reconfiguration of the environment. The reason for this is twofold. First, a change of specification typically involves a change in the environment assumptions, and secondly because in many cases the change in assumptions is due to an actual change of the environment itself. In the case of our motivating example, the change involves unloading the driver for the Polish tool and loading the driver for the Paint tool. Indeed, the old environment assumptions state that the Paint tool is unavailable while the new assumptions state that it is, and explain what can be expected of it. Thus, it is necessary that the controller updates operation control when the change of environment happens, and it will have to signal the change at a point in time where the whole update can be guaranteed to be correct.

We model a controller that is capable of changing its environment dynamically with the term C′′|(E′′ | 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′. Mapping Q, sets the initial state of E′′ according to the current state of E at the time of reconfigure. One possible choice for Q is to map all states of E to one fixed initial state of E′′ modelling fixed initialisation of the environment. In our experience, and in all our case studies, Q was defined to map states in E and E′′ with the same valuations (i.e., vE(ε) = vE′′(ε′′)). For example, if at the time of reconfiguring it is the case that in E Polished(x) is true for some product, it is expected that in the state of E′′ immediately after reconfiguring, the same will hold. Note that we do not require Q to be defined for all states of E, this provides a way of restricting when C′′ is allowed to reconfigure.

On a methodological note, as with hotswap, the alphabet of (E′′ | 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. In addition, note that the term E′′ | reconfigure E′′ assumes that all propositions in the scope of E and not in E′′ (resp. in E′ and not E) are false after (resp. before) reconfigure. Should the update problem require otherwise, E and E′ can be embedded into a larger set of propositions. For instance, E may be used to embed states in E with the valuation of propositions that appear in E′.

An LTKS M is an embedding of M if when restricting the set of propositions of M to those of M they exhibit the same behaviour.

**Definition 4.1. (Embedding)** Let M and M be LTKSs with propositions P_M and P_M. We say that M is an embedding of M into P_M if P_M ⊆ P_M and (M, (P_M, P_M)) ☐ M where ☐ is weak bisimulation [23] where ☐ is defined as N \(→_N \) the LTKS resulting from changing valuation \( ν_N \) in N with \( ν_N \) such that \( p \in ν_N(n) \iff p \in ν(n) \wedge p \not\in X \) for all \( n \in S_N \).

Note that requiring weak bisimulation rather than strong bisimulation allows introducing τ transitions into the embedding. This is essential for modelling situations in which the value for some new propositions are not known to the engineer for a specific state in M. In this case, it is possible to have, for instance, multiple states \{m_i\} that are bisimilar to m in which each \( m_i \) models one possible valuation for the new propositions and can transition via τ transitions to other states in \{m_i\}. A case in which such an embedding is required is discussed in Section 6.2.

4.3 DCU Correctness Criteria

We now formalise the dynamic controller update (DCU) correctness criteria of replacing controller C with C′ assuming the existence of a current specification (in the form of current environment assumptions E, requirements \( G \) with \( G \) a boolean combinations of events, and controllable actions \( A \)), a new specification (similarly, in the form \( E', G' \) with \( G' \) a boolean combinations of events, and \( A' \)), and a transition requirement \( T \) (that constrains the occurrence of events stopOldSpec, startNewSpec and reconfigure). We require as an input to the correctness criteria a specification of how the state of C′ is to be set based on the state of C, i.e. the relation R and also how the state of the reconfigured environment E′ is influenced by the state of E, i.e. the relation Q.

To model the behaviour of updating C with C′ such that the latter can change its environment E with E′ via reconfigure we use the following term: \( (C | hotSwap C') \parallel (E | reconfigure E') \). We require stopOldSpec, startNewSpec and reconfigure 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′, and R should be defined for all states of C. We also require that \( C | hotSwap C' \parallel (E | reconfigure E') \).
to be legal for $E'_{\text{reconfigure}}$ with respect to $A \cup A'$, in other words $C$ and $C'$ never block monitored events ($\ell \notin A \cup A'$).

In the following we define the LTL formula that models the expected behaviour of system $(\langle C_R \mid \text{hotSwap} \rangle, (E_I \mid \text{reconfigure} \rangle) = G_u$:

**Definition 4.2. (Goal for Dynamic Controller Update)** Let $\Box G$ and $\bigcirc G'$ be the current and new goals for a system that is to go from a dynamic update of controllers, and $G$ and $G'$ are boolean combinations of propositions that do not include stopOldSpec, startNewSpec, reconfigure, nor hotSwap. Let $T$ be a 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 LTL formulae:

1. $G \Box \text{stopOldSpec}$
2. $T$
3. $\Box(\text{startNewSpec} \implies \bigcirc G')$
4. $\Box(\text{hotSwap} \implies (\bigcirc \text{stopOldSpec} \land \bigcirc \text{reconfigure} \land \bigcirc \text{startNewSpec}))$

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. Recall that $T$ 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. 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 $G'$. Finally, the last formula requires the controller, once hotSwap occurs, to progress towards the occurrence of events stopOldSpec, startNewSpec and reconfigure.

The following definition puts all the parts together:

**Definition 4.3. (Correctness Criteria for Dynamic Controller Update)** Let $P$ be a tuple $(C, C', G, G', E, E', A, A', T, R, Q)$ where, $C$ and $C'$ are LTKS 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; $(E, \bigcirc G, A)$ and $(E', \bigcirc G', A')$ are the old and new specifications of systems for a dynamic update of controllers, where $G$ and $G'$ are boolean combinations of propositions that neither refer to stopOldSpec, startNewSpec, reconfigure, nor hotSwap; $T$ is an LTL formula modelling the transition requirement, which may refer to stopOldSpec, startNewSpec, and reconfigure, but not to hotSwap; $R$ and $Q$ are mappings between states of LTKS modelling the initial state of $C'$ and $E'$ once they replace $C$ and $E$ at execution time. Let $G_u$ be defined as in Definition 4.2. We say that $P$ is a correct DCU if all the following hold:

1. $(\langle C_R \mid \text{hotSwap} \rangle, (E_I \mid \text{reconfigure} \rangle) = G_u$
2. All states of $C$ are mapped by $R$ to some state in $C'$.
3. $(\langle C_R \mid \text{hotSwap} \rangle, C')$ is legal with respect to $(E_I \mid \text{reconfigure} \rangle)$ on alphabet $A \cup A'$.

Returning to the informal requirements for dynamic controller update discussed in Section 2 rules ii) through v) of that are captured in rules 1 through 4 of Definition 4.2. Rule i) is captured by the fact that hotSwap is not part of the alphabet of $C$, $C'$, $E$ and $E'$, and $R$ is total. Thus, in the term $(\langle C_R \mid \text{hotSwap} \rangle, (E_I \mid \text{reconfigure} \rangle) = G_u$, hotSwap is never constrained.

In this paper we are interested in correct-by-construction approaches to dynamic controller update rather than a construct-then-verify approach. More formally:

**Definition 4.4. (DCU Synthesis Problem)** Let $C$ be a controller for the old specification $(E, \Box G, A)$, and $(E', \bigcirc G', A')$ a new specification, $T$ be a transition requirement, and $Q \subseteq S_E \times S_{E'}$ be a mapping relation.

A solution for the DCU Synthesis Problem is a pair $(C', R)$ such that $(C, C', G, G', E, E', A, A', T, R, Q)$ is a correct dynamic controller update.

5. A SOLUTION

In this section we propose a solution to the synthesis problem formulated in Definition 4.4 based on LTKS control (Definition 3.6). We show how the DCU synthesis problem can be reformulated into a LTKS control problem with specification $(E_u, G_u, A_u)$ such that its solution $C_u$ can be used to build a solution of the form $(C', R)$ of a DCU synthesis problem. We first show how to build $E_u$ and then explain how to extract $R$ and $C'$ from $C_u$.

5.1 Environment Model

The environment model $E_u$ must be built not only to capture $E_{I_{\text{reconfigure}}} E'$ but also to describe how $C_u$ is allowed to use events hotSwap, stopOldSpec, startNewSpec and reconfigure. Furthermore, we shall build $E_u$ in a way that we can later extract from $C_u$ the mapping $R$. To do the latter we shall construct $E_u$ to guarantee that the resulting $C_u$ will have a part of it that is equivalent in terms of behaviour to $C$ up to the occurrence of hotSwap, thus allowing a trivial mapping from states of $C$ to states of $C_u$. In the following, we describe $E_u$ through successive approximations.

To achieve this structural property of $C_u$ we define the environment $E_u$ to initially include the old controller $(C)$ but ensuring that $C_u$ does not control events of $C$ in this first of three phases that $E_u$ will have. When computing a solution for the control problem with specification $(E_u, G_u, A_u)$, $C_u$ will be forced to accept all behaviours of the first phase of $E_u$ because it cannot control any of its events. As a result, $C_u$ will be equivalent in terms of behaviour to that of $E_u$ and $C$ up to the occurrence of hotSwap. Consequently, the first phase of $E_u$ may be defined as $[C]_f$ where $f$ is a relabelling function that maps all events $\ell \in A$ to fresh events that are not in $A'$.

It is when hotSwap occurs that the second phase starts and the update controller must start taking measures that aim to guarantee the correct transition to satisfying the new specification. Thus, the second phase can be described with $E$, while the transition between the phases has the interrupt operator $\xi_{\text{hotswap}}$. However, the problem is how to set the initial state of $E$ after hotSwap. For this we actually need to define the first phase of $E_u$ to be of the form $[E[\ell]]_{f}$ rather than $[C]_f$. This allows defining a relation $I$ that maps states $(c, e)$ of $[E][C]_f$ to states $c$ of $E$. Summarising up to now, the first two phases of $E_u$ can be defined as $[E][C]_f \xi_{\text{hotswap}} E$. 


The third phase is when reconfigure occurs and $E$ is replaced with $E'$ according to the user provided state mapping $Q$. The three phases of $E_u$ can hence be described as $\langle E \parallel Q \cdot (\text{hotSwap}) \parallel E' \rangle$. An informal representation of the three phases of $E_u$ are depicted in Figure 3.

Finally, to ease specification of $G$, $G'$ and $T$, and also to allow a succinct rewrite of $\varphi$ as a safety property $G_u$, we introduce in $E_u$ propositions OldSpecStopped, NewSpecStarted, Reconfigured and HotSwapped that capture the fact that events stopOldSpec, startNewSpec and reconfigure respectively have occurred. This is done by simply composing the three LTKS depicted in Figure 4.

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, $A$ the set of events controlled by $C$, $E$ and $E'$ be the current and new environments, and $Q$ a mapping between $E$ and $E'$ for a dynamic controller update problem. The Environment for the DCU Synthesis Problem ($E_u$) is an LTKS defined as $\langle E \parallel Q \cdot (\text{hotSwap}) \parallel E' \parallel Q \cdot (\text{reconfigure}) \parallel E'' \rangle$, where $I$ is a mapping from states of the form $(c, e)$ of $E\parallel Q$ to states of the form $(c', e')$ of $E\parallel Q$ such that $I(e) = f$ is a function that relabels all events $\ell \in A$ to fresh events $\ell'$.

**5.2 Solving DCU synthesis with LTKS control**

We can now solve the DCU synthesis problem by defining an LTKS control problem from which a controller $C'$ and a relation $R$ can be extracted.

The LTKS control problem can be stated as in Definition 5.1 using an environment $E_u$ as defined in Definition 5.1, the goal $G_u$ as defined in Definition 4.2, and the set of controlled events $A_u$ as defined as the union of the controlled events $A$, $A'$ and $\{\text{stopOldSpec, startNewSpec, reconfigure}\}$.

The solution $C_u$ to the DCU LTKS problem with specification $(E_u, G_u, A_u)$ can be shown to have a set of states, all reachable from the initial state, that are bisimilar to $\langle C \parallel F \rangle$ up to the occurrence of hotSwap (recall $F$ from Definition 5.1).

This follows straightforwardly from the fact that $E_u$ initially behaves as $\langle E \parallel F \rangle$ and that $C_u$ cannot restrict its behaviour due to the relabelling function $f$. Indeed, $C_u$ will have three phases, the first in which it behaves as $C$, the second in which it is handling the transition between specifications (dealing with $\cup G$, $\cup G'$ and $T$ and reconfiguration) and the third in which both stopOldSpec and startNewSpec have occurred and $C_u$ is simply dedicated to satisfying $G'$ (see Figure 5).

The first phase allows defining $R$ from states of $C$ to those of $C_u$ straightforwardly: if $c$ in $C$ is bisimilar to $c_u$ in $C_u$ up to hotSwap and $(c_u, \text{hotSwap}, c')$ is a transition of $C_u$, then $R$ should map $c_u$ to $c'$. Note that $c_u$ will always have a hotSwap transition enables as $I$ covers all states in $\langle C \parallel F \rangle$.

Having defined $R$, we now construct $C'$ from $C_u$. Note that $R$ maps states all states in $C$ to states the second phase of $C_u$ (see $C_T$ in Figure 6). This means that states in the first phase of $C_u$ will never be visited in term $C_T \parallel R \parallel \text{hotSwap} C_u$.

Thus, we construct $C'$ to be the portion of $C_u$ that does not includes states from its first phase. This ensures that $C'$ does not have a complete replica of $C$ within it, avoiding controller bloating (particularly if several controller updates are chained over time).

**Theorem 5.1.** Let $S$ be a DCU synthesis problem with specification $(G, G', E, E', A, A', T, Q)$, $E_u$ be defined as in Definition 5.1, and $A_u = A \cup A'$.

If $C_u$ is the solution to the LTKS control problem with specification $(E_u, G_u, A_u)$, $R$ a binary relation such that $(c, c_u) \in R$ if and only if $c_u \in C_u \parallel (t, \text{hotSwap}, c') \in \Delta C_u$ and $c$ is similar to $\langle [C_u]_{\text{cut}}(c_u) \rangle$, where cut$(\ell) = \ell \Leftrightarrow \ell \neq \text{hotSwap}$, and $C'$ is the result of removing any state in $C_u$ that can reach hotSwap, then $(C', R)$ is a solution for $S$. In addition, if $(C', R)$ is a solution for $S$, then there is a solution for the DCU control problem with specification $(E_u, G_u, A_u)$.

Proof of the theorem above is straightforward because of the similar structures of $G_u$ and $G'$ and the fact that $E_u$ is built to mimic $E'$. The key part of the proof relies on showing that the first phase of $C_u$, up to hotSwap is similar to $C$. This is proven based on the fact that $C$ is a memoryless controller with respect to $E$ and $G$ and that $G_u$ does not require the new controller to have any additional memory up to hotSwap. This means that $C_u$ is also memoryless (up to hotSwap) with respect to $E$, $C$ and $G_u$. Finally as the first phase of $E_u$ is built with all events in $C$ as uncontrollable, then the first phase of $C_u$ is guaranteed to include all of $C$.

**5.3 Complexity**
Solving an LTKS control problem (as defined in Definition 3.6 in Section 3) for any LTL property is 2EXPTIME complete \[10\]. 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, \( \bigwedge_{i=1}^{n}(DS_i \lor \diamond R_i) \)). LTKS 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. 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 \[17\] to the DCU synthesis problem when the transition requirement \( (T) \) is a safety property. The case studies described below were resolved using this tool. Both case studies and tool are available at \[1\].

6. VALIDATION

The purpose of our validation is to show applicability of the approach by resolving case studies taken from literature and also the generality of the approach with respect to existing work.

All case studies were run using an extension of the MTSA tool \[17\]. MTSA natively supports specification of LTS \[27\] and properties using a textual, process algebraic, notation 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 defining to automatically compute \( A_u, G_u \) and \( E_u \), and solve the DCU control problem. We used fluents to implicitly define LTS as LTKS. The extended version of the tool and case studies can be found at \[1\].

We selected the Rail Cab \[20\], Power Plant \[36\], GSM-oriented protocol \[53\] and MetaSocket \[52\] case studies to allow comparison with the work closest to ours. We selected a dynamic workflow update case from \[19\] and the Production Cell \[31\] to illustrate how the current limitation of workflow systems technology could be overcome some of its limitations (currently limited to requiring quiescence before updating, e.g., \[52\]). Finally, we chose a UAV setting aimed showing an end-to-end application of the technique, from synthesis to enactment on an adaptive system infrastructure.

For each case study we fixed old and new specifications for the update and explored the use of various transition requirements. We used two default transition requirements on all case studies in addition to domain specific ones. We used a non restricting transition requirement \( (T_\forall = \top) \) which allows dropping the old specification at any point and also a period in which anything is allowed before starting the new specification. We also used the requirement that prohibits events to occur during the period in which neither specification holds \( (T_\exists = \neg\text{NewSpecStarted} \land \neg\text{OldSpecStopped}) \implies \neg\text{Event} \), where \( \text{Event} \) is a disjunction of all propositions indicating the occurrence of events.

Overall, more than 20 DCU synthesis problems were defined and solved, corresponding to different choices of transition requirements for each case study, were synthesised and analysed. The majority of the controllers were synthesised in a few seconds. Noteworthy examples of computationally more complex controllers were those that had more elaborate \( T \) requirements for the Production Cell and the UAV case studies. These took up to 5 minutes of computation time, for environments of up to 15000 states, yielding controllers of up to 10000 states.

6.1 Power Plant

In \[36\] a controller for the cooling system of a nuclear power plant is discussed. The current controller is required to service maintenance requests by first stopping the cooling agent pump and then restarting it. The new controller specification is not required to stop and start the pump. There is also a system invariant which states that the cooling agent pump may not be stopped indefinitely (this may lead to a devastating accident). The authors show that if an update is performed at a state in which the current controller has stopped the pump but not restarted it, then the plan risks an accident as the new controller may not restart the pump. A safe way of preserving the system invariant 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 resolved three different DCU control problems for this case study. Both with \( T_\forall \) and \( T_\exists \) the system exhibits the invalid behaviour described in \[36\]. We also used \( T = ((\text{OldSpecStopped} \land \neg\text{NewSpecStarted}) \implies (\text{endProcedure} \implies \neg\text{PumpOff})) \) 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 off-line update" requirement in \[20, 36\]. Not only does \( T \) allows updating specifications in strictly more states than in \[20\] while being correct with respect to \( \varphi \), it also avoids the manual correctness validation required in \[36\] for weaker transition requirements than \[20\]. Maximum synthesis processing time was negligible (under 0.1 seconds), while the environment models and controllers size were under 100.

6.2 RailCab

The RailCab system \[34\] consists of autonomous vehicles must coordinate to transport passengers and goods on demand. The subsystem discussed in \[20\] focuses on controlling RailCabs as they approach a crossing. The RailCab can monitor events such as \text{endOfTrunkSection} \ and that it has passed the \text{lastBrake} \ or \text{lastEmergencyBrake} \ opportunity. It controls the \text{brake} \ and \text{emergencyBrake} \ also can receive responses to queries it controls such as \text{requestPermissionToEnter}. The goal for the current and new controller requires ensuring that the RailCab enters the crossing only if it has been granted permission to do so. There are constraints on when permission may be requested, brakes applied and also assumptions on when responses to controlled actions happen.

The difficulty of the update scenario is that the new specification introduces a new milestone (\text{approachingCrossing}) and further constraints related to it. This requires embedding the current environment description with appropriate
propositions (e.g., PassedApproachCrossing). However, as the current controller is not monitoring if it has passed this milestone, the value of such proposition is unknown in $\hat{E}$. This is modelled by refining each state in $E$ to two states in $\hat{E}$ (one where the proposition is true and the other false).

We resolved three DCU control problems for three different transition requirements. For $T_r$, the controller exhibits unsafe behaviour. Using $T_0$ the resulting controller can perform a safe update in more states than the one in $[20]$: if an update is requested while the RailCab is between endOfTrunkSection and lastBrake. In $[20]$ the update will be postponed until after lastBrake as there is uncertainty as to whether approachingCrossing occurred. In the controller we synthesise, such non-deterministic behaviour is handled automatically assuming the worst (the milestone was reached, but not sensed) but acting opportunistically should the event be sensed later.

Note that in addition, the controller for $T_0$ will then, as in $[20]$, postpone startNewSpec if lastEmergencyBrake has occurred. This is because, according to the new requirements, it would be a violation to do checkCrossingStatus so late and it would also be a violation to enter the crossing without having asked checkCrossingStatus. This is a contradiction which can only be avoided by sticking to the old specification. Such a postponement may be considered undesirable by some stakeholders: If it is too late to checkCrossingStatus, have the Railcab brake just in case. This requirement is inconsistent with both the old and new specifications for the Railcab, as both only allow braking if negative responses to either checkCrossingStatus or requestPermissionToEnter are received. Thus, we built a controller that allows a transition period in which neither specification holds, but when the RailCab is in the very last track section (after lastEmergencyBrake) we require an emergencyBrake. The transition property is formalised to include: $T = (\text{OldSpecStopped} \land \neg\text{NewSpecStarted}) \implies ((\neg\text{return} \lor \text{brake}) \implies \text{PassedLastEmergencyBrake})$.

### 6.3 Production Cell

For the production cell setting discussed in Section 2 which is based on that of [21] we modelled various transition requirement alternatives and built update controllers for them. In addition to the transition requirement discussed in Section 2 we required that the new specification be put in place when the production line has no products being processed (startNewSpec $\implies$ Empty). Interestingly, with this requirement update cannot be guaranteed: As $\text{in}(x)$ is not controllable, it is impossible to guarantee that the production line will be empty at some point. We also forced the system to always work under some specification ($T_0$) which allows the cell to drop the old specification at any time as long as it can continue processing any partially processed product without violating the new specification. Another alternative we modelled was to require that any partially processed product (i.e., has had some tool applied to it) be finished with the old specification ($T = \text{ToolApplied} \implies G W \text{startNewSpec}$) or be finished with the new specification ($T = \text{ToolApplied} \implies G' W \text{startNewSpec}$).

### 6.4 Other case studies

Due to lack of space we do not describe the rest of the case studies we ran. Succinctly, we synthesised solutions to the problem of changing workflows at runtime to respond to changes in business goals as described in [19] for a billing and shipping workflow. This represents a typical challenge in the area [19, 50]. We replicated the GSM-oriented protocol update from [53], showing how the manually built solution can be automatically synthesised with our technique using an appropriate $T$ requirement. We also produced update controllers automatically for the MetaSocket case study in [52]. As in [52] modelled the different update scenarios and performed various chained updates in which we confirmed that our technique does not suffer from controller (or DCU synthesis problem) blowing. Finally, we modelled a surveillance scenario in which a UAV running a mission has its mission goals and onboard capabilities changed while in the air. For variants of this case study not only we compute different update controllers but we also enacted them on a real UAV system using the framework for adaptive systems [11] that follows the rationale of the reactive concentric control structure in [18] and is a simplified instance of the MORPH reference architecture [10].

The models for these four case studies and the ones described previously in the Section can be found at [4].

### 7. DISCUSSION AND RELATED WORK

The problem of dynamic 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 pursued objective of the update (see [4] for a survey). The bulk of the effort in dynamic update assumes that either there is no specification change and hence the same behaviour (minus bugs to be fixed) is expected (e.g., [26]), or the specification is generic (e.g., [12, 35, 3, 24, 29] and not user provided. Examples of the latter, apart from ensuring the update does not lead to crashes, are ensuring type safety (e.g., [47]) and data isolation between versions [46]. Quiescence [29] and related notions (e.g., [51, 3, 24]) do not originally deal with an explicit representation of the properties to be preserved, but have been used in conjunction with techniques that ensure generic semantic consistency (e.g., [7]).

The need for user-specified update properties has been recognised in [30, 8, 25, 20, 36, 53, 42]. In [25], specification of update properties is considered but the focus is on verification rather than synthesis. Our work builds on [52] who identify the need for different update patterns (e.g., one-point and overlapped specifications) but also account for situations in which neither the old nor new specification holds and also allows scope change. In contrast, [20, 36] have a fixed notion of correctness, not supporting user specified criteria for transition requirements. Although Zhang and Cheng [53] also 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. [32] build on this semi-automatic approach a tool capable of selecting and applying the best adaptation safe path that balances non-functional requirements, based on cost values. Our work can be extended to consider non-functional requirements by using quantitative control synthesis techniques such as [9].

As in [20, 36], we consider use of synthesis to update a controller in a reactive system. However, 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. This notion of assuring liveness (the update eventually happens) is a key distinguishing feature of our approach. Although in [20] the liveness problem is acknowledged, the strategy of how to guide the current system to a safe updatable state is user-provided.

Section 6 compares our approach with [20] and [36] via case studies. To the best of our knowledge, these are the first and only papers that investigate automated resolution of dynamic controller update in which there is a specification change explicitly provided. They adopt a very natural and general correctness criteria which relieves the engineer of specifying transition requirements (in contrast to our approach) but at the cost of limiting the kind of updates that can be supported. In [20], 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 [20] can be expressed as a transition requirement in this approach, but additionally, progress towards update can be guaranteed in our approach. In [36], weakened update criteria with respect to that of [20] 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 5).

Synthesis has been used extensively to guarantee code that is correct by construction (e.g., [23]). The fully automated nature of synthesis naturally leads to applying it 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 [27] the problem of evolving component assemblies is addressed by synthesising glue code (i.e. controllers).

An assumption of this work is that it is possible to dynamically reconfigure the software components that the controller is bound to for receiving environment events and actuating on it’s environment. This is consistent with much of the work on adaptation such as [5, 19, 18]. Furthermore, we assume it is the controller that can trigger the reconfiguration (as in [10]). In other words, that if the new specification introduces different environment characteristics (e.g., a new component, a change in the call protocol of an existing component, a disabled component), it is the controller that can decide when these changes occur (i.e. the controller controls some middleware infrastructure that can load/unload/change components at runtime). This allows the controller to plan a change and gives more freedom to find a strategy that can satisfy the change in requirements.

In many situations, an unannounced change in the environment can occur and updating the controller to accommodate this change is desirable. 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 [13] 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 [15] 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.

In this presentation we have not permitted the inclusion of liveness goals as part of the current and new specifications. This makes presentation simpler and also allows for a linear resolution complexity of the DCU control problem if the control environment is deterministic. Nonetheless, it is possible to allow further expressiveness in $G$, $G'$ and $T$ without incurring in the full penalty of solving control problems (2EXP TIME-COMPLETE). It is possible, for instance to reformulate Definition 12 to allow specifications $G$, $G'$ and $T$ to include subformulas of the form $\varnothing \varphi$. Such a bichu acceptance criteria extends significantly expressiveness while remaining in a polynomial time complexity.

8. CONCLUSIONS

We have shown how the problem of dynamically updating a controller to satisfy a new specification can be posed as a control problem. The proposed solution guarantees satisfaction of the new specification and any relevant transition requirements that may be given. Furthermore, 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 and satisfy the new specification.

Future work involves looking to increase the expressive- ness of goals to liveness without having to pay the full 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.

9. REFERENCES


