Assured Mission Adaptation of UAVs

SEBASTIÁN A. ZUDAIRE, Instituto Balseiro - Universidad Nacional de Cuyo, Argentina
LEANDRO NAHABEDIAN, Universidad de Buenos Aires / CONICET, Argentina
SEBASTIÁN UCHITEL, Universidad de Buenos Aires, Argentina and Imperial College London, United Kingdom

The design of systems that can change their behaviour to account for scenarios that were not foreseen at design time remains an open challenge. In this paper we propose an approach for adaptation of mobile robot missions that is not constrained to a predefined set of mission evolutions. We implement an adaptive software architecture and show how controller synthesis can be used both to guarantee correct transitioning from the old to the new mission goals with runtime architectural reconfiguration to include new software actuators and sensors if necessary. The architecture brings together architectural concepts that are commonplace in robotics such as temporal planning, discrete, hybrid and continuous control layers together with architectural concepts from adaptive systems such as runtime models and runtime synthesis. We validate the architecture flying several missions taken from the robotic literature for different real and simulated UAVs.

Additional Key Words and Phrases: Dynamic Controller Update, UAV Applications, Discrete Event Controller Synthesis, Cyberphysical Systems.

ACM Reference Format:
Sebastián A. Zudaire, Leandro Nahabedian, and Sebastián Uchitel. 2022. Assured Mission Adaptation of UAVs. *J. ACM* 37, 4 (January 2022), 27 pages.

1 Introduction

Adaptive systems are capable of changing their behaviour while running in response to changes in their environment, capabilities and goals [26]. Adaptation can be addressed at various levels of abstraction to respond to many different kinds of changes. In this paper we focus on mobile robot adaptations that involve responding to unforeseen changes in the high-level mission goals that a Unmanned Aerial Vehicles (UAVs) must achieve. This scenario typically involves a human-in-the-loop that must define what the new mission goals are and when should they be deployed.

Adapting mission goals at runtime puts forward a number of challenges. It is desirable to change the behaviour of the system as soon as possible, but taking into account that the UAV is flying and performing tasks that if interrupted carelessly may result in unsafe behaviour. Stopping a UAV (e.g., hovering, landing) at a safe mission state to make any necessary adaptations to start pursuing the new mission goals is not always possible or desirable. Furthermore, a strategy defined for updating the behaviour of the UAV based on the old mission, the current state and the new mission may not be a valid strategy once all relevant software is transmitted and uploaded onto the vehicle, bound

Authors' addresses: Sebastián A. Zudaire, sebastian.zudaire@ib.edu.ar, Instituto Balseiro - Universidad Nacional de Cuyo, Río Negro, Argentina; Leandro Nahabedian, lnahabedian@dc.uba.ar, Universidad de Buenos Aires / CONICET, Buenos Aires, Argentina; Sebastián Uchitel, suchitel@dc.uba.ar, Universidad de Buenos Aires, Buenos Aires, Argentina and Imperial College London, United Kingdom.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
0004-5411/2022/1-ART $15.00
https://doi.org/

into the software architecture and in place to take control of the new UAV mission: the vehicle and environment conditions may have changed since the start of the process.

In this paper we explore the question of how UAV systems can be designed to support adapting to unforeseen circumstances by changing missions at fly-time. Our hypothesis is that discrete event controller synthesis at runtime can help provide a flexible mission adaptation mechanism with guarantees not only about satisfying the new mission requirements but also safely transitioning between the current and the new mission requirements.

Consider a UAV that is performing a remote patrol mission, flying at a high altitude between series of patrol points, recording the ground with a camera to relay a low-res movie through a low-bandwidth channel back to a control centre. While in mission, a report comes into the control centre that a person, known to be wearing a red jacket, has gone missing near the patrol area.

Rather than flying the UAV to base, programming a search mission for the UAV, deploying new software and sending the UAV back to the original patrol area, it would be convenient to have the UAV designed to support the following scenario.

While in flight, personnel at the control centre specify a new mission for the UAV that involves systematically searching the area at low altitude using a red-sensor detection filter on high-res photos, and landing on detection. The team prepares for upload an image processing module. They also specify key mission transition requirements: the module cannot be bound into the software while the camera is in use and the camera must be realigned to point downwards before the new mission is started.

Having produced the specifications and new software modules, mission command pushes a button and code is automatically synthesised to satisfy the transition requirements and the new mission. The synthesised code is uploaded onto the UAV and starts running. The current mission is stopped, the camera realigned and set to high-res mode, the new flight altitude is configured and then, ensuring that the camera is not in use, the uploaded image processing module is bound into the architecture. Once the UAV reaches the new altitude, the search mission commences.

The mission adaptation should be *assured* in the sense that the there are guarantees that the adaptive system will (a) drop the old mission, reconfigure the software architecture, and start the new mission correctly with respect to the transition requirements and (b) the new mission will be started and the system behaviour will be correct with respect to the new mission goals as long as specified assumptions on the underlying software infrastructure and physical environment hold. The problem of assured dynamic change in software systems is a key concept in correct mission adaptation and has been recognised by many [10, 11, 18, 22, 46–48, 56, 57, 69, 71, 77, 80, 83, 86, 96].

We report on a robotic system that supports assured runtime adaptation of missions. The mission (specified as a combination of automata and temporal logic formulae) can involve re-discretization of the robot workspace and reconfiguration of the robot's software architecture, with changes in the available software sensors and actuators. Correctness criteria for the mission transition and reconfiguration is provided by the user as a temporal logic formula. The system relies on discrete event-controller synthesis to produce a plan that safely reconfigures and transitions into the new mission, providing the *assurance* through a correct-by-construction update strategy. The plan is executed on a hybrid control architecture that supports runtime swapping of plans, and runtime binding and unbinding of hybrid components. To the best of our knowledge this is the first robotic system that implements runtime synthesis for assured mission adaptation.

The main contribution of this paper is a system for adapting UAV missions with correctness guarantees that (a) builds on discrete event controller update [69] but extends it to liveness properties to support typical mobile robot missions [67], (b) that implements a hybrid controller [59] architecture that incorporates reconfiguration capabilities from [17]. We demonstrate, in real and

simulated flights, how a UAV running a mission can be adapted at runtime to new missions that may require changes to workspace discretization, software sensors and software actuators.

We begin by introducing the preliminaries in Section 2 including a synthesis method for dynamic update of controllers for safety goals. In Section 3 we present an extension to liveness goals of discrete event controller update. In Section 4 we show how UAV missions and mission adaptations can be specified. This sets the appropriate level of abstraction to present in Section 5 the software architecture and main software components we use to implement assured adaptation of UAV missions. We then report on our validation efforts in Section 6 and conclude with a discussion on results and related work in Section 7 and conclusions.

2 Preliminaries

2.1 Labelled Transition Systems

Labelled Transition Systems (LTS) [54] are a common representation of reactive systems. LTS are automata where transitions are labelled with events that constitute the interactions or communication between concurrent processes, each described as an LTS. We use LTS to model the assumptions about the behaviour of a cyber-physical system that is to be controlled by a discrete event controller. We also use LTS to represent such discrete event controllers. We partition events into controlled and uncontrolled to specify assumptions about the environment and safety requirements for a controller.

Definition 2.1. *(*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 $e_0 \in S_E$ 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 $(e, \ell, e') \in \Delta_E$. We say that a sequence of events $\pi = \ell_0, \ell_1, \ldots \ell_k$ is a trace of E if there are k states such that $(e_i, \ell_i, e_{i+1}) \in \Delta_E$, with $i \in \{1, \ldots, k\}$. The notion of trace can be extended to infinite traces straightforwardly.*

Complex models can be constructed by LTS composition. We use a standard definition of *parallel composition* ($\|$) that models the asynchronous execution of LTS, interleaving non-shared actions and forcing synchronisation of shared actions.

Definition 2.2. *(*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:*

- *if $(e, \ell, e') \in \Delta_E \wedge \ell \notin A_C$ then $((e, c), \ell, (e', c)) \in \Delta_\|$,*
- *if $(c, \ell, c') \in \Delta_C \wedge \ell \notin A_E$ then $((e, c), \ell, (e, c')) \in \Delta_\|$, and*
- *if $(e, \ell, e') \in \Delta_E \wedge (c, \ell, c') \in \Delta_C \wedge \ell \in A_E \cap A_C$ then $((e, c), \ell, (e', c')) \in \Delta_\|$*

We use an *interrupt operator* [69] ($E \lightning_f^{\ell} E'$) to model that the behaviour described by LTS E may be interrupted by event ℓ to become LTS E'. Function f sets the initial state of E' based on the state of E when the interrupt happens.

Definition 2.3. (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 $E \lightning_H^{\alpha} N$ 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:*

- *if $(e, \ell, e') \in \Delta_E$ then $(e, \ell, e') \in \Delta_\lightning$,*
- *if $(n, \ell, n') \in \Delta_N$ then $(n, \ell, n') \in \Delta_\lightning$, and*
- *if $(e, n) \in H$ then $(e, \alpha, n) \in \Delta_\lightning$*

2.2 Fluent Linear Temporal Logic (FLTL)

Linear temporal logics are commonly used to describe system goals . We use a linear temporal logic of fluents (FLTL) [43] that allows referring to abstract states interpreted over sequences of events. A fluent $fl = \langle Set_\top, Set_\bot, v \rangle$ is defined by a set of initiating actions ($Set_\top$), a set of terminating actions ($Set_\bot$), and an initial value v true ($\top$) or false ($\bot$). We may omit set notation for singletons and use an action label ℓ for the fluent defined as $fl = \langle \ell, Act \setminus \{\ell\}, \bot \rangle$, where Act is the universe of all observable events. Thus, the fluent ℓ is only true just after the occurrence of the action ℓ, until any other action occurs.

Let Π be the set of infinite traces over Act. The trace $\pi = \ell_0, \ell_1, \ldots$ satisfies a fluent $fl = \langle Set_\top, Set_\bot, v \rangle$ at position i, denoted $\pi, i \models f$, if and only if, one of the following conditions holds:

- $v \wedge (\forall j \in \mathbb{N} \cdot 0 \leq j \leq i \Rightarrow \ell_j \notin Set_\bot)$
- $\exists j \in \mathbb{N} \cdot (j \leq i \wedge \ell_j \in Set_\top) \wedge (\forall k \in \mathbb{N} \cdot j < k \leq i \Rightarrow \ell_k \notin Set_\bot)$

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.

Let $\mathcal{F}$ be the set of all possible fluents over Act. A FLTL formula is defined inductively using the standard Boolean connectives and temporal operators **X** (next), **U** (strong until) as follows: $\varphi ::= f \mid \neg\varphi \mid \varphi \vee \psi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \psi$, where $f \in \mathcal{F}$.

Given an infinite trace π, the satisfaction of a formula φ at position i denoted $\pi, i \models \varphi$, extends the satisfaction relation for fluents as follows:

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

We say that φ holds in π denoted $\pi \models \varphi$, if $\pi, 0 \models \varphi$. We say that φ holds for LTS E, denoted $E \models \varphi$, if for every trace π of E we have $\pi \models \varphi$. We say φ is a safety formula if every trace that does not satisfy it has a finite prefix such that, however it is extended to an infinite path, it still violates φ. All other are refered to as liveness formulae.

Boolean connectives $\wedge$ and $\rightarrow$ are defined as usual over $\neg$ and $\vee$. In addition, temporal connectives are interpreted as usual [8]: $\Diamond\varphi = \top \mathbf{U} \varphi$, $\Box\varphi = \neg\Diamond\neg\varphi$, and $\varphi \, \mathrm{W} \, \psi = (\varphi \mathbf{U} \psi) \vee \Box\varphi$.

2.3 Discrete Event Controller Synthesis

We adopt the controller synthesis formulation from [30]. Given an LTS E describing the execution environment of a discrete event controller with a set of controllable actions L and a task specification φ expressed in FLTL, the goal of discrete event controller synthesis is to find a discrete event controller modelled as an LTS C such that $E \| C$: (a) is deadlock free, (b) C does not block any non-controlled actions, and (c) $E \| C \models \varphi$. We say that a control problem $\langle E, \varphi, L \rangle$ is *realizable* if such an LTS C exists. The tractability of the controller synthesis depends on the size of the problem (i.e., states of E and size of φ) and also on the fragment of the logic used for φ. When goals are restricted to safety formulae and Generalised Reactivity (1) (GR(1)) formulae the control problem can be solved in polynomial time [74]. GR(1) formulas are of the form $\bigwedge_{i=1}^{n} \Box\Diamond A_i \implies \bigwedge_{i=1}^{m} \Box\Diamond G_i$ where A_i and G_i are Boolean combinations of fluents. This allows writing goals for controllers such as $\Box\Diamond\text{request} \implies \Box\Diamond\text{response}$.

2.4 MTSA

We use MTSA [31] for modelling and solving control problems. MTSA provides a graphical environment in which LTS can be built using a simple text-based language (FSP), analysed via FLTL

model checking and also used as input to a variety of control problems. Both model checking and controller synthesis use an explicit state-based representation of LTS. For controller synthesis, two implementations are provided. The first implements a fixpoint algorithm to compute a state ranking [52, 53] over the composed environment model of the control problem. The second adds an on-the-fly approach to avoid building the entire environment model should it be expressed as the parallel composition of multiple LTS.

MTSA supports validation of the environment LTS, the FLTL goals and assumptions, and the resulting controller through graphical animation features, abstraction via hiding and minimisation, (bi-)simulation checks, and model checking among other features.

2.5 Hybrid Controllers

To execute discrete event controllers on a robot system, a hybrid controller architecture as described in [59] is commonly used [13]. Hybrid controllers are used to translate controllable events such as $\text{go}.l_i$ into low-level continuous movement that guide the robot to the physical location modelled by l_i. Similarly, the hybrid controller generates uncontrollable events by monitoring low-level sensor data to, for example, generate event $\text{at}.l_i$ when the robot arrives at the physical location denoted by l_i. Ultimately, the hybrid controller implements an interface that matches the controller's set of controlled and uncontrolled events, hiding the physical continuous world and exposing it as discrete events.

This interface is derived from the continuous to discrete abstraction process [59]. An important step of the process is selecting an adequate *discretization* for the robot's workspace (e.g., grid-based [94], triangulation [13]). As in the rest of the abstraction, it is user-provided, but several tools exist to automate this discretization process (e.g., [36]).

Hybrid control architectures may have different layers and layout [13, 27, 51, 59, 72]. In this work we will build on a three layer interface as explained in [72, 97] which consist of: (a) a *Discrete Event Layer* which executes the discrete event controllers, (b) a *Hybrid Control Layer* which produces the translation between high-level events and low-level movement and actions, (c) a *Robot Layer* which contains the feedback-controllers and other capabilities of the robot system. In terms of discrete event controller synthesis, the *execution environment* modelled with an LTS E captures the behaviour of the *Hybrid Control Layer* together with the *Robot Layer* and all the continuous physical dynamics involved.

2.6 Dynamic Controller Update (DCU)

The Dynamic Controller Update (DCU) problem [69] addresses the problem of changing the goals of a system that is being controlled by a discrete event controller without stopping the system. In [69] the DCU problem is formalised and a solution proposed under the restriction that the current controller resulted from a control problem for a safety goal and that the new goal for the system is also a safety property.

Four events that are specific to DCU problems are introduced: the uncontrolled event hotSwap that denotes the instant in which the current controller is replaced by the controller to be synthesised, the controlled events stopOld and startNew that are expected to signal the moment from which the original system goal is no longer guaranteed and the instant from which the new goal is expected to be ensured, and the controlled event reconfig that dynamically reconfigures the hybrid control and robot layers.

A DCU problem assumes a controller C that is solution to a control problem $\langle E, \varphi, L\rangle$ where $\varphi = \Box G$ is a safety goal. It also assumes a new safety goal φ', a safety update requirement Θ which cannot refer to hotSwap and an LTS $\hat{E}$ that models how the environment will change upon a reconfig

command. $\hat{E}$ must exhibit the same behaviour as E up to the occurrence of reconfig. Note that it may be convenient to build $\hat{E}$ compositionally from a model E of the current environment, a model E' of the environment once reconfigured and a mapping g from E to E': $E \lightning_g^{\text{reconfig}} E'$.

The safety update requirement Θ typically constrains the occurrence of reconfig, startNew and stopOld to indicate when it is safe to reconfigure the environment, when it is safe to drop the old safety requirements and any particular safety constraints that must hold between dropping the old requirements and starting to enforce the new ones. An example is $\Theta_\emptyset = \Box(\neg \mathit{OldStopped} \vee \mathit{NewStarted})$, with fluents *OldStopped* and *NewStarted* turning on with the occurrence of stopOld and startNew, respectively, and never turning off. We will refer to this domain independent formula $\Theta_\emptyset$ as the *standard transition requirement*. $\Theta_\emptyset$ states that the system must always be under control to achieve the old specification (φ) or the new one (φ').

A solution to the DCU problem is an LTS C' and a function f such that (a) f is a total function from the states of C to those of C', (b) $C \lightning_f^{\text{hotSwap}} C'$ does not block any non-controlled actions in $\hat{E}$, and (c) $(C \lightning_f^{\text{hotSwap}} C') \| \hat{E}$ is deadlock free and satisfies φ_{DCU} defined as:

$$\Box(\text{hotSwap} \implies (\Diamond\text{reconfig} \wedge \Diamond\text{stopOld} \wedge \Diamond\text{startNew})) \wedge (G \text{ W } \text{stopOld}) \wedge \Theta \wedge \Box(\text{startNew} \implies \varphi')$$

In other words a solution is a new controller C' and a function f for setting its initial state based on the current state of the current controller C. We refer to a solution (C', f) as an *update strategy*. The overall behaviour of applying the strategy, i.e., hotswapping C with C' according to f, modelled by $C \lightning_f^{\text{hotSwap}} C'$ should ensure that once the hotswap occurs, then the environment must be reconfigured, the old specification dropped and the new one adopted $C \lightning_f^{\text{hotSwap}} C' \models \varphi_{DCU}$.

In [69], the DCU problem is reduced to a standard GR(1) discrete event control problem and is solved within the MTSA [31] tool.

2.7 Iterator-Based Hybrid Control

Controller synthesis may suffer from state-explosion as the size of the environment E can grow exponentially in the number of LTS composed in parallel. This means that even with a polynomial synthesis procedure (e.g., when using the GR(1) fragment), for certain control problems with large environment, synthesising a controller may not be possible. This is particularly the case for mobile robot mission scenarios like UAV mapping [84] or search and rescue [28] which have potentially large discrete workspaces. If each location is represented explicitly as in [34, 94] then the number of locations in which the UAV can be grows and introduces a combinatorial explosion of the overall environment states.

An alternative approach is presented in [97]: iterator-based planning proposes the use of a hybrid control architecture that has a data type for storing discretized locations that make up the robot's workspace. The hybrid layer exposes a call (has.next?) that advances the iterator and returns if any locations remain. Locations can be removed (remove.next) and the iterator reset (reset). The hybrid layer also implements go.next which commands the robot to go to the location that is currently selected by the iterator, and the corresponding at.next. An iterator in combination with a sensor-based planning approach [61] that introduces capabilities for sensing, for example, if the current location selected by the iterator is to be visited, allows expressing mission requirements that can be synthesised in constant time with respect to the number of locations in the workspace. This is because the controller does not deal with locations explicitly, it simply requests locations one at a time, senses what kind of locations they are, and acts accordingly.

3 Dynamic Controller Update of Live Missions

Changing the goals of a robotic system mid-mission is a challenging task. In particular, providing assurances regarding system behaviour while transitioning between the current robot goals and the new ones is important as there may be teardown activities for the current goal and setup activities for the new goal that include not only constraints on behaviour but also changes in the software architecture.

In [69] the problem of changing goals for systems that are controlled using discrete event controllers is described as a dynamic controller update (DCU) problem. The new goals (φ'), constraints on the transition phase Θ and a description of the expected environment behaviour after reconfiguration $\hat{E}$ are used to build a new controller C' and an initialisation function f that allow hotswapping an existing controller C to ensure correct teardown, reconfiguration, and startup during the transition (Θ), and from then on the new goals (φ').

The DCU problem in [69] is restricted to safety requirements, i.e., both the current system goal (φ) and the new system goal (φ') must be safety properties. However, a typical mobile robot mission will have liveness properties.

Safety properties specify things that should never happen (e.g., the UAV should never enter the no-fly zone: $\Box\neg\mathsf{at.NoFlyZone}$). Liveness properties allow specifying good things that should eventually happen. For instance, and taking patterns recollected from scientific literature and industrial case studies [67], patrolling two locations requires $\Box\Diamond\mathsf{at.1} \wedge \Box\Diamond\mathsf{at.2}$ and a delivery mission will require a property such as $\Box(\mathsf{available} \implies \Diamond\mathsf{deliver})$. A strategy for realizing a mission specified only with safety properties may be simply not taking off to avoid violating any safety property.

Taking the definition of φ_{DCU} from the DCU problem in [69] and simply allowing φ to be a liveness property is not appropriate: the conjunct of φ_{DCU} that refers to the current goal is of the form (G W $\mathsf{stopOld}$) assuming that $\varphi = \Box G$. This states that the controller can stop ensuring G once it has signalled $\mathsf{stopOld}$ (which may have constraints attached, based on Θ). Replacing G with a liveness formula leads to a problem: consider $\varphi = \Box(\mathsf{available} \implies \Diamond\mathsf{deliver})$. In this case φ_{DCU} will include the conjunct $(\mathsf{available} \implies \Diamond\mathsf{deliver})$ W $\mathsf{stopOld}$. Such a requirement is too restrictive as it does not allow scenarios where a user may want to specify urgent updates in which pending deliveries can be aborted.

To provide a more general framework for updating controllers, we redefine the property φ_{DCU} that $(C \lightning_f^{\mathsf{hotSwap}} C') \| \hat{E}$ is expected to satisfy. We assume without loss of generality that the current mission goals (φ) are split into safety ($\varphi_S = \Box G$ a safety property) and a liveness part (φ_L). With this partitioning of mission goals, it is possible to rephrase the requirement on termination of the old specification only in terms of its safety part and no conditions of ensuring its liveness part after $\mathsf{stopOld}$.

In summary, we define an update requirement that supports liveness φ_{DCU_L} as:

$$\Box(\mathsf{hotSwap} \implies (\Diamond\mathsf{stopOld} \wedge \Diamond\mathsf{reconfig} \wedge \Diamond\mathsf{startNew})) \wedge (G \text{ W } \mathsf{stopOld}) \wedge \Theta \wedge \Box\big(\mathsf{startNew} \implies \varphi'\big)$$

Note that the new update requirement φ_{DCU_L} drops preserving any liveness obligation of the old specification. Thus, no live behaviour is required as soon as the new controller is put in place. Should this not be desired, the user can include in Θ requirements regarding how to treat any pending obligations. For instance in the case where the old specification requires $\varphi = \Box(\mathsf{available} \implies \Diamond\mathsf{deliver})$, a user may use the transition requirement to state that the old specification may only be dropped when there are no pending deliveries: $\Theta = \Box(\mathsf{available} \implies (\neg\mathsf{stopOld} W \mathsf{deliver}))$. Alternatively, a user may specify that any pending deliveries must be honoured before the system can be reconfigured but not necessarily before dropping the old specification: $\Theta = \Box(\mathsf{available} \implies (\neg\mathsf{reconfig} W \mathsf{deliver}))$.

Although this new formulation of DCU allows for arbitrary changes of goals, the synthesis of a controller may be prohibitively expensive as full FLTL synthesis is required (which is double exponential). However, if φ' is constrained to safety (φ'_S) plus liveness formulae of the form $\bigwedge_{i=1}^{m} \Box\Diamond G'_i$ then φ_{DCU_L} can be expressed as a safety plus GR(1) formula:

$$\Big(\Box\Diamond \mathit{HotSwap} \implies \Box\Diamond(\mathit{OldStopped} \wedge \mathit{NewStarted} \wedge \mathit{Reconfigured}) \wedge \bigwedge_{i=1}^{m} \Box\Diamond G'_i\Big)$$
$$\wedge\ (G\ \mathrm{W}\ \mathsf{stopOld})\ \wedge\ \Theta \wedge\ \Box(\mathsf{startNew} \implies \varphi'_S)$$

where *OldStopped*, *NewStarted*, and *Reconfigured* are fluents initially false and become true once stopOld, startNew and reconfig occur.

Thus, the dynamic controller update problem in [69] can be adapted to support update of missions with live goals and in particular it can solved in polynomial time if the new mission goals are restricted to recurrent liveness goals of the form $\bigwedge_{i=1}^{m} \Box\Diamond G'_i$. We have extended the implementation of the MTSA [31] tool to support solving these control problems (available at [99]).

4 Specification and Synthesis of Assured Adaptation Plans

In this section we show how assured mission adaptation of mobile robots can be framed as a Dynamic Controller Update (DCU) problem. A solution to the DCU problem yields a controller that codifies a plan that ensures the mobile robot will change its current mission plan to a plan that satisfies its new mission. In the next section we discuss a hybrid control architecture that can make use of a solution to the DCU problem to actually adapt a UAV at runtime.

We first show how a simple case mobile robot mission adaptation can be solved using DCU. We illustrate the importance of obtaining one controller that works for any reachable state of the current mission plan (i.e., $C \lightning_f^{\mathsf{hotSwap}} C'$). Nonetheless, this example is simple in that the term $E \lightning_g^{\mathsf{reconfig}} E'$ of the DCU control problem allows $E = E'$ and g be the identity function. The transition requirement Θ also plays a minor role.

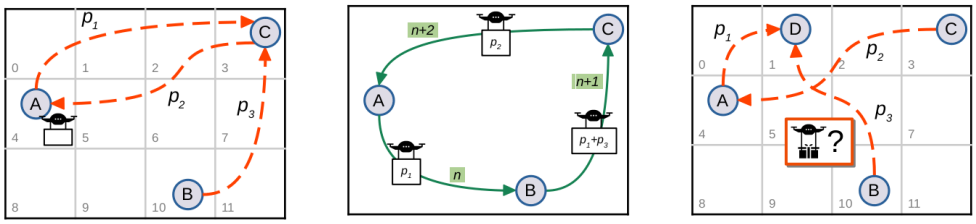
The second example discusses a mission adaptation that requires introducing new software components and re-discretization of the robot's workspace. For this, the DCU problem requires $E \neq E'$ and an appropriate mapping function g. The third example, shows the relevance of Θ to solve adaptation problems in which the new and old missions are logically inconsistent.

4.1 Adaptation of Live Missions

The mission examples will be taken from the pickup and delivery family of problems [87], where goods or passengers have to be transported from different origins to different destinations. In particular they belong to the *one-to-one* subclass in which the aim is to design vehicle routes in order to satisfy a set of pickup and delivery requests between location pairs, subject to side constraints [24].

Example 1 (Delivery Service): consider a UAV operating as a delivery messenger between three discrete locations **A**, **B**, and **C**, transporting package types p_1, p_2 and p_3 between them, assuming there is a limitless amount of packets at each node. In Fig.1a we depict the pick-up and delivery requirements for each package type. Additionally, it is required that the UAV must not move between locations without a package to preserve a minimum weight requirement. Assume the UAV is executing a plan depicted in Fig. 1b that satisfies these requirements, travelling between **A**, **B**, and **C**, in that order, moving packages.

Assume that at some point, while the UAV is flying, the mission needs to be updated to incorporate a new location **D** and different delivery requirements as depicted in Fig. 1c. Note that the location of the UAV is marked as unknown in Fig. 1c as the UAV is constantly moving while the



(a) Original delivery requirements and initial UAV location.

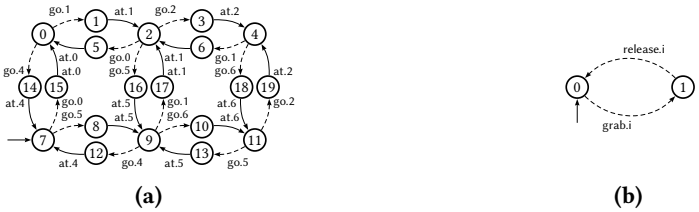
(b) Original delivery plan, showing the UAV correctly delivering each of the packets.

(c) New delivery requirements with new drop off location **D**, initial UAV position unknown.

Fig. 1. Package delivery. Plan in (b) is shown schematically without the discretized regions. Red arrows indicate required source and target of package type delivery. Green arrows are planned legs, labels indicate the order in which they occur ($n = 0 \ldots$ indicates loop number).

requirements are being defined (and eventually deployed). The requirement of non-empty flights is maintained.

The original mission plan can be synthesised by defining a discrete abstraction for the workspace of the robot and constraining robot movements to adjacent cells. In Fig. 2a we show a portion of the LTS covering only cells 0−2, 4−6, where we model the movement actions as control modes [9] with a controllable (go.i) and uncontrollable (at.i) pair. We also model the grab and release mechanisms for the three package types p_1, p_2, p_3 as controllable actions using the LTS in Fig. 2b. Note that the initial location of the UAV is modelled by the initial state of the LTS of Fig. 2a.



(a) **(b)**

Fig. 2. (a) Movement restrictions in the discretized workspace (snippet). (b) Grab and release model for package p_i, with $i = 1, 2, 3$.

We now formalise the mission goals, specifying where each package type can be grabbed and released. For instance, we define a safety property $\varphi_{p_1} = \Box\big((\text{grab.1} \implies At.4) \wedge (\text{release.1} \implies At.3)\big)$ to require packages of type p_1 be taken from **A** to **C**. Fluents $At.i = \langle\{\text{at.i}\}, \{\text{go.k} \cdot 0 \leq k \leq 11\}, \bot\rangle$ are true when the robot is at location i, with $0 \leq i \leq 11$ as the number of total locations the UAV can be in is 12. Additionally we require $\psi_{p_1} = \Box\big(Carrying.1 \wedge At.3 \implies (\neg Moving \text{ W } \text{release.1})\big)$ to ensure that the UAV will deposit packets as soon as it arrives at the respective target location, with fluents *Moving* and *Carrying.i* being turned on/off with go/at actions and when the UAV does a grab/release of package p_i, respectively. Avoiding empty trips is accomplished by adding another safety specification: $\gamma = \Box(Moving \implies Carrying.1 \vee Carrying.2 \vee Carrying.3)$.

Finally, we add the liveness property of continuously delivering packages p_1, p_2, p_3: $\rho = \Box\Diamond\text{release.1} \wedge \Box\Diamond\text{release.2} \wedge \Box\Diamond\text{release.3}$. We refer to $\rho \wedge \gamma \wedge \bigwedge_{i:1,\ldots,3} \varphi_{p_i} \wedge \psi_{p_i}$ as OldSpec.

Note that the mission specification assumes the availability of an infinite number of packages to be delivered. This can be weakened to allow for finite packages of each kind with events that indicate package availability. We avoid this for presentation purposes.

A controller C for this mission can be automatically built (as discussed in Section 2.3) by providing the specification (OldSpec) and an environment E (the parallel composition of the LTS in Fig. 2). Note that OldSpec can be rewritten as a combination of safety properties and a GR(1) property. The resulting controller (using MTSA) exhibits the following trace: grab.1, go.8, at.8, go.9, at.9, go.10, at.10, grab.3, go.11, at.11, go.7, at.7, go.3, at.3, release.3, release.1, grab.2, go.2, ... A graphical depiction of the UAV being controlled is given in Fig. 1b.

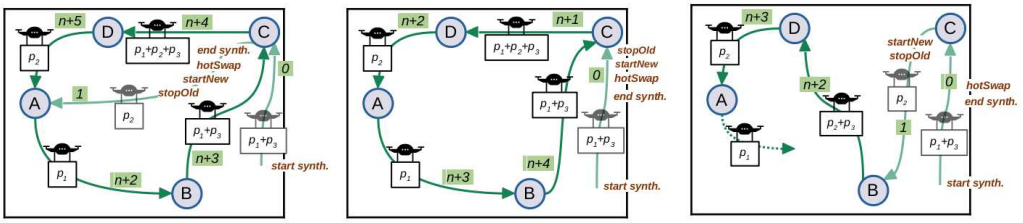
We now discuss adapting the mission plan to achieve delivery requirements of Fig. 1c. Note that there is no change in the discretization of the workspace and the functioning of the grab/release actuation modes. Thus, we can reuse the LTS models of Fig. 2 to define the environment model for the new mission plan (i.e., $E = E'$). The FLTL properties φ_{p_i} and ψ_{p_i} must be changed slightly to reflect the new delivery relations shown in Fig. 1c. Assume these properties to be φ'_{p_i} and ψ'_{p_i}. We refer to $\rho \wedge \gamma \wedge \bigwedge_{i:1,\ldots,3} \varphi'_{p_i} \wedge \psi'_{p_i}$ as NewSpec.

To perform a mission update we must formulate a DCU problem which requires not only the old and new mission specifications but also two further inputs: a function g from E states to E' states is required and a transition property Θ constraining (if needed) the occurrence of stopOld, startNew, and reconfig.

Given that $E = E'$, we define g as the identity function. This means that the new controller when in place will assume that the current state of the environment E' is the same as the current environment of E. Or more precisely, at the occurrence of the reconfig event, the execution environment of the controller can be assumed to behave as E' setting its initial state based on the current state of E.

For this simple example, it suffices to use the *standard transition requirement* $\Theta_\emptyset = \Box(\neg \mathit{OldStopped} \vee \mathit{NewStarted})$ mentioned in Section 2.6 requiring the system to be satisfying one of the two mission requirements. We will discuss more complex transition requirements in the next examples.

Having defined NewSpec, E', g and Θ, and given that the current mobile robot is running a controller C to achieve OldSpec in environment E, we have a fully formulated DCU problem (with live missions, see Section 3) for which a solution (C and f) can be constructed.



(a) Update plan when the hotSwap occurs between **C** and **A**, while carrying p_2.

(b) Update plan when the hotSwap occurs between **B** and **C**, while carrying p_1 and p_3.

(c) Partial view of the update plan with weight requirements, showing delayed update when hotSwap occurs in the same position as (d).

Fig. 3. Package delivery. Plans in (d-f) are shown schematically without the discretized regions. Green arrows are planned legs, labels indicate the order in which they occur ($n = 0 \ldots$ indicates loop number). We omitted the occurrence of the event reconfig since in these scenarios reconfiguration is trivial.

Consider the scenarios in Fig. 3a and 3b in which the UAV controlled by C is flying towards location **C** carrying two packages as part of its plan to satisfy OldSpec when a synthesis procedure to find a solution to the DCU problem is run. The scenarios differ in when the synthesis procedure ends (see *end synth.*)

In Fig. 3a, while C' is being computed the UAV reaches location **C** and, as per OldSpec drops off package p_1 and p_3, picks up p_2 and then continues towards location **A** where it must deliver p_2. On flight towards **A**, the DCU synthesis procedure ends, the new controller C' to be hotswapped in and its current state is set in terms of the current state of C and function f. At this point C' declares that NewSpec will hold from now on (startNew) and that OldSpec will not be guaranteed anymore (stopOld). It does so in this order to comply to Θ. Function f has been computed to preserve the state of C in C', thus C' "knows" that it is carrying p_2 and is on its way to **A**. From then on C' commands the UAV as per NewSpec.

In Fig. 3b, C' and f are computed before reaching **C**. This time controller C' is hotswapped in before the UAV reaches **C** and its initial state is set differently by f than before because C is in a different state. Now C' "knows" that it is carrying p_1 and p_3 and is on its way to **C**. The new controller declares startNew and stopOld and upon reaching **C** it can no longer (as in the previous scenario) drop p_1 and p_3 as it would be inconsistent with NewSpec. Instead, it picks up p_2 and continues pickups and drop offs as per NewSpec.

Note that when the computation of C' and f started, no assumption is made as to whether the computation will end before or after the UAV reaches location **C**. Thus, it is the same C' and f that are computed in both scenarios. This demonstrates the need for C' to have a strategy for transitioning into the new mission that works for any state in which the current system might be in. It is f at hotswapping time that determines which state C' should be set at, and consequently which transition strategy should be used.

In the two previous scenarios, the mission switch was performed immediately after hotswapping controllers. This is not always the case. Consider a scenario in which the user introduces into the NewSpec an additional requirement forbidding transportation of three packages (to avoid overstraining the UAV): $\Box\neg(\mathit{Carrying.1} \wedge \mathit{Carrying.2} \wedge \mathit{Carrying.3})$. Assume that similarly to Fig. 3b the computation of C' and f terminates before the UAV reaches location **C** (see Fig. 3c). Here, the new controller cannot immediately start satisfying NewSpec as picking up p_2 would violate the requirements. Hence, the synthesised update controller chooses to *delay* the change of specification, first dropping off p_1 and p_3, and also picking up p_2 as required in OldSpec. Only then, it switches mission and flies to **B** rather than **A**.

Note that we have deliberately omitted referring to the occurrence of event reconfig for simplicity. We discuss this event in subsequent scenarios.

4.2 Dealing with Re-Discretization and new Capabilities

The simple example from the previous section avoided a key difficulty in real mission adaptation: *what happens when the new mission requires or must deal with a change in the execution environment of the robot?* By execution environment we refer to hardware that may be malfunctioning, software with new sensor or actuating capabilities that must be uploaded, or changes in the assumptions that are considered valid given the conditions of the physical world in which the robot is operating. Ultimately, from a control perspective all these changes represent modifications in the set of controllable and non-controllable events, the formulation of the environment LTS E' and mission goals φ'.

Consider the following example:

Example 2 (Reconfiguring Delivery Service): assume the original mission specification from *Example 1.* A new pick-up/drop-off location **D** must be introduced. The location falls beyond the

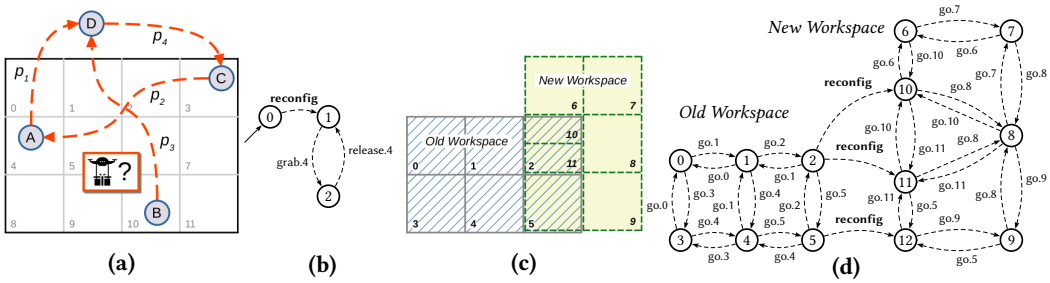


Fig. 4. (a) Variant of updated delivery mission. (b) Reconfiguration model for packet p_4. (c) Simplified scenario of workspace reconfiguration. (d) Reconfiguration model for the workspace in (c), with all the at.i transitions removed for clarity.

current discretized workspace. In addition, a new package type p_4 is to be transported. The delivery requirements are shown in Fig. 4a. The non-empty trips requirement is kept. The distinctive shape of package type p_4 requires a software module tailored specifically to control the robot's gripper to successfully pick them up and drop them off.

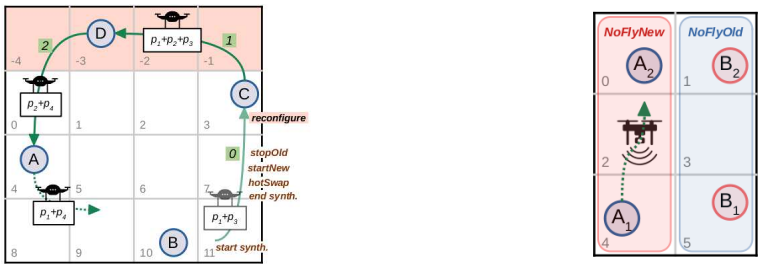
The DCU control problem uses the reconfig to model the change in the execution environment of the robot. For this example, two reconfiguration aspects need to be modelled by the user. First, a model for the p_4 grab/release module must be introduced, ensuring that its initial state is one in which no p_4 package is being held (see Figure 4b). The second, is the reconfiguration of the workspace discretization.

To analyse how a workspace discretization may be reconfigured we use an alternative simplified workspace change depicted in Fig. 4c . Here some of the discrete cells are only present in the old workspace (0, 1, 3, and 4), some are only present in the new workspace (6–9), cell 5 is present in both and there is a change of granularity for cell 2, which now maps to 10 and 11. We model the mapping between the states of the old and new environments in Fig. 4d. The reconfiguration is required to happen when the robot is in one of the shared discrete cells (2 and 5), where the choice of where cell 2 maps is non-deterministic from the controllers perspective. This requirement arises from the desire of the user to ensure that the UAV is always in a location within the workspace of the current mission. To ensure this it must be in a region belonging to the intersection of both workspaces in order to proceed with the reconfiguration.

Composing the LTS in Fig. 4b for p_4, with the models for $p_1, \ldots, p_3$ (see Fig. 2b) and one along the lines of Fig. 4d (but now considering the original workspace discretization problem in Fig. 4a), generates a model for $E \not\downarrow_g^{\text{reconfig}} E'$ (instead of providing E' and g separately).

The new delivery requirements, modelled according to Fig. 4a, must be included in NewSpec, and a transition requirement Θ must be provided. We assume the simple $\Theta_\emptyset$ used previously that requires the UAV to always be constrained according to either of the missions.

The resulting DCU problem can be solved and a new controller C' and controller initialization function f can be computed for the *Example 2* scenario. We show in Fig. 5a an update scenario that the new controller may exhibit. Synthesis starts and ends with the UAV on its way to location **C**. The new controller C' is hotswapped in and upon arriving to location 3 (at.3), the controller commands a reconfiguration. This is possible because location 3 is part of the old and new discretized workspace. With reconfig, the UAV infrastructure is changed: a new module for grabbing and releasing p_4 packages is added to the software architecture. The UAV is then commanded to location **D** via newly introduced discrete locations $(-1, \ldots, -4)$. At -3, the new module is used to pick up a p_4 package. Control proceeds satisfying the new mission requirements.



(a) (b)

Fig. 5. (a) Partial view of the update plan for the scenario in Fig. 4a, where hotSwap occurs between B and C. (b) Inconsistent patrol mission update.

4.3 Inconsistent Mission Adaptations

Sometimes, the behaviour of a new mission may be logically inconsistent with the current UAV mission. For these cases, the transition requirement $\Theta_\emptyset = \Box(\neg \mathit{OldStopped} \vee \mathit{NewStarted})$ which we used in the previous scenarios is not adequate: if the two missions are logically inconsistent, there is no safe state in which to first do startNew and then stopOld. Moreover, failure to synthesise a controller for the DCU problem using $\Theta_\emptyset$ means that such inconsistency exists (assuming the new specification is realizable). In these scenarios, we rely on the user to provide an adequate domain-dependent transition requirement for mission adaptation as we illustrate in a simple example.

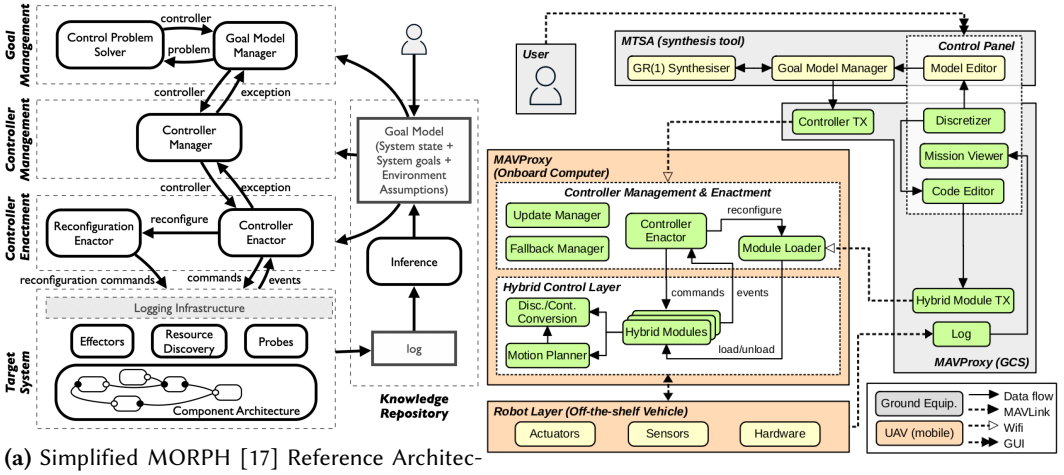
Example 3 (Surveillance Update): consider a typical UAV patrol mission as described in [67] for surveillance of two areas A_1 and A_2 as shown in Fig. 5b. To restrict the movement area of the UAV the user imposes an area **NoFlyOld** as a no-fly zone allowing other vehicles or humans to work in this region. The user now decides that the surveillance must now be done between areas B_1 and B_2, and moves the no-fly zone to **NoFlyNew**, as shown in Fig. 5b.

The original UAV mission goal can be written as: $(\Box\Diamond \mathit{At.0} \wedge \Box\Diamond \mathit{At.4}) \wedge (\Box\neg \mathit{At.}\boldsymbol{\mathit{NoFlyOld}})$. That is, be at cells 0 and 4 infinitely often and never be within the **NoFlyOld** region. Similarly, the new mission can be specified as $(\Box\Diamond \mathit{At.3} \wedge \Box\Diamond \mathit{At.5}) \wedge (\Box\neg \mathit{At.}\boldsymbol{\mathit{NoFlyNew}})$, using appropriately defined fluents.

Note that in this example if we used the transition requirement $\Theta_\emptyset$ the DCU problem has no solution, as is not possible to switch from the old goal (OldSpec) to the new goal (NewSpec) without violating one of them. If the UAV is in locations from the left column (0, 2, 4) then it cannot switch to achieving NewSpec because these locations are in the new no-fly area. If the UAV is moved to locations from the right column to comply to NewSpec then it is violating OldSpec. This is an extreme example that motivates the need for specifying requirements that deal with *transitioning behaviour* between missions.

A trivial but unsatisfactory solution to resolving inconsistencies is to impose no transition requirements $\Theta = \top$. However, this allows arbitrary behaviour: the controller may declare stopOld, and once relieved of following the old mission requirements perform arbitrary actions before performing startNew. Note that the latter must eventually occur as $\Diamond$startNew is required.

To resolve the inconsistency between the old and new no-fly zones in the patrol mission change, a reasonable transition requirement may be to allow a period in which neither old nor new mission restrictions are satisfied but restrict what can occur during this period. For instance, to restrict movement between no-fly zones to the bottom side of the grid. That is, when the old specification



(a) Simplified MORPH [17] Reference Architecture

(b) Architecture for Assured Mission Adaptation of UAVs

Fig. 6. Adaptive Architectures

is dropped, the UAV must be at locations 4 or 5 until the new one is adopted:

$$\Theta = \Box\Big(\mathit{OldStopped} \implies \big((\mathit{At.4} \vee \mathit{At.5})\ \mathrm{W}\ \mathit{NewStarted}\big)\Big) \tag{1}$$

With this transition requirement, it is possible to synthesise a controller that satisfies the mission adaptation.

5 Adaptive Architecture and Implementation

In the previous section we showed how to cast assured mission adaptation as a DCU problem. In this section we will show how a solution to a DCU problem can be used in a robotic system to effectively provide assured mission adaptation. To this end, we build on the notion of hybrid controller [61] to address three implementation challenges: (a) **uploading and hotswapping** new discrete controllers at runtime, (b) loading and unloading of software components to allow **coordinated software reconfiguration**, (c) **human-in-the-loop support** for runtime specification of mission adaptations. We address these challenges by taking elements from the MORPH [17] reference architecture and integrating them with hybrid controllers. MORPH (Figure 6a) outlines a framework for architectural adaptation through the runtime synthesis, hotswapping and enactment of correct-by-construction strategies.

Firstly, we explain how MORPH and hybrid architectures resolve the three implementation challenges while providing links to the concrete architecture we implemented (Figure 6b). Secondly, we provide implementation details of the architecture that supports assured mission adaptation of UAVs.

5.1 Architectural Extensions for Adaptability

Hybrid controllers serve as an interface (*Hybrid Control Layer*) between the discrete high-level events of the synthesised controllers (*Discrete Event Layer*) and the low-level sensors, feedback-controllers and other actuators from a robot (*Robot Layer*). As a result, these architectures help produce continuous movement and trajectories that satisfy the user specification. In terms of MORPH, the *Hybrid Control Layer* and *Robot Layer* live in the *Target System* (see Figure 6b), while the continuous execution of the discrete event controller (*Discrete Event Layer*) is what occurs within the

Controller Enactor component (see Figure 6a). The MORPH *Effectors* map to *Actuators*, while *Probes* map to *Sensors*, together with the required *Hybrid Modules* to make them work. To implement the rest of the components from the MORPH architecture we included several other modules (not present in the hybrid control architectures from [72, 97]) that we will describe next.

MORPH proposes dealing with controller **uploading and hotswapping** as follows: the *Controller Management Layer* is responsible for replacing the controller currently being enacted by a new one. This can be triggered by the reception of a new controller from the *Goal Management Layer* or due to an exception raised by the *Controller Enactor* (the *Controller Management Layer* may store fallback controllers). Note that MORPH does not provide any guidance or mechanisms to ensure that the hotswapping is correct, neither does it define correctness for that matter.

In our implementation, the MORPH *Controller Management Layer* is a component (*Update Manager*) that uses the output of MTSA synthesis tool for a DCU control problem to hotswap the current controller in the *Controller Enactor*. To do so, it must take the MTSA output, $C \lightning_f^{\text{hotSwap}} C'$ where C is the controller currently running in the *Controller Enactor*, extract C' and f, identify the current state of C, hotswap C with C' within the *Controller Enactor* and set the state of C' according to f. It must do this procedure atomically. In other words, the *Update Manager* receives information that maps every possible state of the current controller C with a state in the new controller C'. Upon reception it atomically replaces C with C' setting the current state of C' according to the current state of C.

Our implementation also includes a *Fallback Manager* that provides a preset fallback discrete event controller that is to be used if an event is received that is not enabled in the current state of the controller being enacted.

The MORPH *Goal Management* layer's responsibility is to produce controllers for the *Controller Management* layer. It constructs control problems based on a *Knowledge Repository* and uses a *Control Problem Solver* to produce discrete event controllers. In our implementation, the MTSA tool (see top of Fig. 6b) implements both the *Goal Management* layer and part of the *Knowledge Repository* by providing functionality for representing knowledge of the robots capabilities, environment assumptions and mission goals, a GR(1) synthesis procedure and transformation procedures for various control problems (including DCU) to GR(1). The implementation also includes the *Controller TX* module for uploading the result of MTSA to the robot.

In MORPH, **software reconfiguration** is also considered. Note that in Figure 6a a simplified version of software reconfiguration is depicted, one in which it is assumed to be atomic; this is not always the case. The *Controller Enactor* commands a *Reconfiguration Enactor* to reconfigure and the latter then reconfigures the *Target System*. How new software modules are loaded is unspecified in MORPH.

In the concrete architecture we developed, reconfigurations are limited to basically adding and removing hybrid modules that implement abstract events and commands that may appear in a discrete event controller (e.g., new capabilities, a different *Motion Planner*) and modules for discrete to continuous conversion of discrete locations (i.e., allowing re-discretization). New modules (and their mapping to events/actions) are received and stored by the *Module Loader*. Instructions for unloading unnecessary modules can also be received. Upon reception of the reconfig command the *Module Loader* loads and unloads the corresponding modules into the *Hybrid Control Layer*, and changes the mapping between events and methods calls to incorporate the controllable events implemented by the new modules. The robot processing (e.g., flight control) occurs oblivious to this reconfiguration as all communication between the *Robot Layer* and *Hybrid Control Layer* occurs through a non-blocking protocol.

MORPH prescribes **human-in-the-loop support** for adaptation via a *Knowledge Repository*. This repository accumulates knowledge from logged data from the *Target System* and combines it with user supplied input to close the adaptation loop. How this information is specified, inferred and stored is not prescribed by MORPH, however it is assumed that all elements required for deriving adaptation strategies are provided via the *Knowledge Repository*.

Specifics of the tools used to implement the user interaction and the *Knowledge Repository* are provided in the next section. The main components however are:

- *Model Editor:* text editor to specify the original and update synthesis problems using LTS models and FLTL formulae.
- *Discretizer:* interactive map that allows the user to draw regions and set the granularity at which they are to be partitioned into discrete locations. The output of this component are location names that can be referred to in the LTS models and FLTL formulae via the *Model Editor*, and a hybrid module written in Python that will be loaded onto the UAV to perform continuous location - discrete location transformations.
- *Mission Viewer:* collection of graphical representations of the data from the *Log* that provide the user significant input about the ongoing mission.
- *Code Editor:* standard code editor that allows the user to program the new software modules required during the adaptation. The model E' introduced by the user through the *Model Editor* must correctly model the behaviour of the new software modules that are programmed here and later uploaded during the adaptation.

5.2 Implementation on UAVs

We now discuss implementation specifics of an architecture for assured mission adaptation of UAVs, including the main architectural hardware components and a key robotics software package we used: MAVProxy. Our working fork of MAVProxy to support hybrid control of discrete event controllers on UAVs as well as assured mission adaptation can be found in [98].

The system (see Fig. 6b) comprises of three main hardware components: an *Off-the-shelf Vehicle*, an *Onboard Computer* that runs the adaptation software, and a general purpose computer that acts as a *Ground Control Station* (GCS).

The *Off-the-shelf Vehicle* includes hardware required for flying (propellers, rudders, batteries, motors, etc.) and an embedded processor that runs communications software supporting MAVLink [5], a lightweight messaging protocol for receiving commands and sending telemetry. The processor also runs software for its sensors and actuators including various feedback-control loops (often referred as Autopilot or Flight Controllers) that implements MAVLink commands such as commanding the vehicle to navigate to a specified waypoint, calibrating sensors, return-to-launch emergency commands, and arming and disarming the vehicle. In our experimentation we used a Parrot Ar.Drone 2.0 [16, 21], a simple quadcopter, that has proprietary communication with basic MAVLink capabilities. We also used the ArduPilot Software-In-The-Loop (SITL) UAV simulator as in [7, 25]. The SITL simulator allows us to test UAV systems loaded with ArduPilot firmware (e.g., custom made as in [78] or commercial as the 3DR Solo Drone [7]) without the UAV hardware. This simulator (see [97]) has been used to seamlessly go from testing to actually flying a custom made fixed-wing vehicle (e.g., [84]) based on a Pixhawk (e.g., [68, 81]).

Similarly to [23, 33, 92] we expand the computing capabilities on the vehicle by physically fixing on top a general purpose *Onboard Computer* that runs most of the adaptive software architecture. We use a Raspberry Pi 3B+ single board computer when simulating with the ArduPilot SITL (to emulate realistic computing capabilities for an onboard computer), and a lighter Raspberry Pi Zero W mounted on the vehicle when flying the Parrot Ar.Drone 2.0.

Finally, for the *Ground Control Station*, a standard laptop computer is needed to run the discrete event controller synthesis software [31]. We used a Linux laptop with an Intel i7 3.5GHz processor and 12GB of RAM.

A key element of the architecture is the MAVProxy software package [85]. This is a widely used (e.g., [14, 23, 68, 85]) GNU GPL Python package for UAVs that implements the MAVLink protocol. The package includes many standard modules, from firmware management to a camera viewer and a moving map, that allow configuring a MAVProxy process to support different vehicle setups and mission tasks.

MAVProxy is designed to be used as ground control software (e.g.,[68]). That is, MAVProxy running on a computer on the ground and providing a series of modules that provide mission monitoring capabilities and high-level commands for setting up and running missions. However, since MAVProxy provides a simple mechanism using custom modules for ad-hoc extensions, it has also been used onboard the vehicle (e.g., [23]) to provide new functionality.

The architecture runs a lean MAVProxy instance on the *Onboard Computer* including only default standard modules that connect via WIFI or serial communication with the embedded processor on the *Off-the-shelf Vehicle*. However, we add a number of custom modules to implement *Controller Management & Enactment*, and *Hybrid Control* layers.

Note that the system conformed by the *Onboard Computer* running the MAVProxy instance and the *Vehicle* running the flight controllers is a fully autonomous vehicle that does not need communication to a ground computer to fly its mission.

The *Ground Control Station* runs a separate MAVProxy instance, configured similarly to most MAVProxy uses. It communicates directly with the vehicle to perform the initial mission setup, to receive telemetry and to allow taking control over the vehicle if necessary. Telemetry data is shown on a GUI to users using standard modules. This ground MAVProxy instance communicates with the airborne MAVProxy instance using a custom protocol over TCP/IP via WIFI to support the interactions between the *Controller Management* layer and the *Synthesiser*. The *Control Panel* is conformed with joint elements from MAVProxy and MTSA, which provides functionality for specifying discrete event control problems as described in Section 2 and provides a back-end *Synthesiser* layer.

As a summary we show in Fig. 7 the inputs that must be provided by a user to update the mission of a UAV that is running on our architecture. Original and update mission specifications (input for MTSA) are created from LTS and FLTL formulas defined by the user, potentially referring to a workspace discretization generated by the *Discretizer* from user-defined *Regions of Interest*. The output controllers, the hybrid module responsible of continuous-discrete location transformation (constructed by the *Discretizer*), and *New Software Capabilities* written by the user are then loaded onto the UAV through the transmission modules.

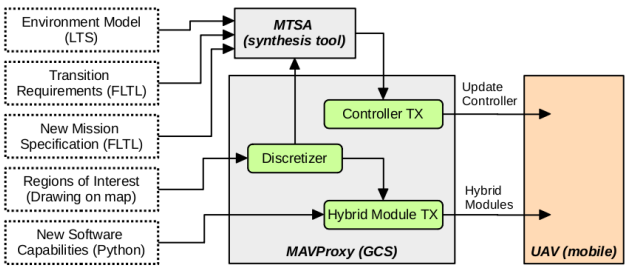
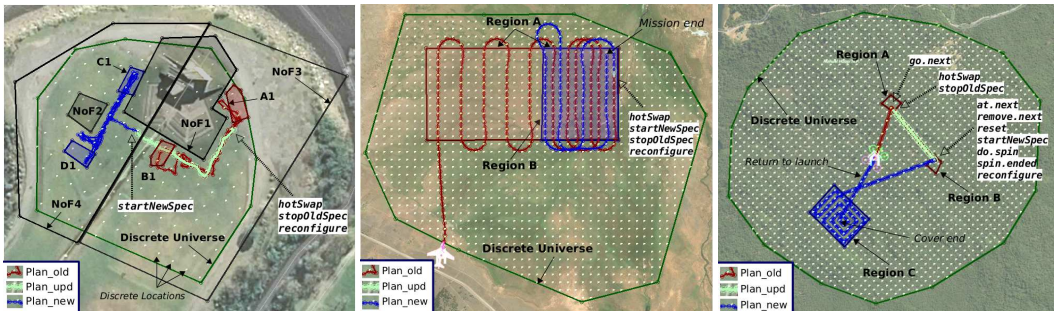


Fig. 7. User Provided Inputs for Assured Mission Adaptation. (Architectural abstraction of Fig. 6b).



(a) (b) (c)

Fig. 8. (a) Parrot Ar.Drone 2.0 real flight path with an unexpected goal change. (b) ArduPlane simulation for a mission degradation scenario. (c) ArduCopter simulation for a fire lookout and cover mission.

6 Validation

In this section we report on various flights and adaptations we ran to validate our approach. We aimed to validate various characteristics of our adaptive system. Namely we pursued:

- *Feasibility* by running multiple missions and informally validating that the resulting UAV behaviour is consistent with the intended behaviour.
- *UAV flexibility* by using different UAVs. We flew a Parrot Ar.Drone in Section 6.1 and 6.4, and ArduPilot simulations for fixed-wing UAVs (ArduPlane) and quadcopters (ArduCopter) in Sections 6.2 and 6.3, respectively.
- *Hybrid Control Layer flexibility* by implementing two different abstraction approaches: Iterator-Based Planning (Sections 6.2 and 6.3) and Explicit-Location Planning (Sections 6.1 and 6.4).
- *Mission variability* by studying different missions types with varying discrete universe sizes, ranging from 48 to 1834 discrete locations. Missions patterns we used are common in the literature (see [67] for a survey). Within this item, we also looked at the ability of the system to support non-trivial *reconfiguration* by introducing various types of new sensors.

The videos and specifications for the simulated and real flights can be found in [99].

6.1 Unexpected Goal Change

We revisit the Example 3 of Section 4.3. The original mission consists of a typical patrol mission as in [27, 38, 64, 67, 93] for surveillance of two areas **A1** and **B1** with two no-fly zones: **NoF1** to avoid a local obstacle in the fly region and **NoF4** as the **NoFlyOld**. These areas are shown in Figure 8a. For this mission we used discrete cells of 10 m × 10 m and a flight height of 1.5 m, with a universe of 163 discrete locations.

A discrete event controller can be synthesised using a similar explicit-location abstraction as in Figure 2, expanding the model to include takeoff and landing events, fluents defined as in Section 4.1, a safety rule $\square\neg\mathsf{land}$ to avoid unnecessary landing and the following requirement: $(\square\neg At.NoF1) \wedge (\square\neg At.NoF4) \wedge (\square\diamond At.A1) \wedge (\square\diamond At.B1)$.

The resulting control problem size was 320 states and a controller was synthesised in 0.5 s using up to 20.1 MB of RAM and automatically loaded onto the Parrot Ar.Drone 2.0 which started the mission and produced the trajectory indicated as *Plan_old* in Figure 8a. While flying the *Plan_old*, we specified a new goal: two new areas to be patrolled **C1** and **D1**, together with the no-fly regions **NoF2** due to local obstacles and **NoF3** as the **NoFlyNew**. To avoid inconsistency between the two

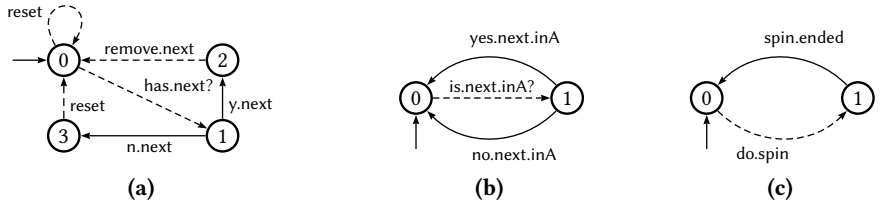


Fig. 9. (a) Iterator LTS. (b) Sensor A LTS. (c) Spin capability LTS.

missions, we added a transition requirement similar to (1) but prohibiting the local obstacles **NoF1** and **NoF2** instead of forcing the UAV to be in locations 4 or 5.

The dynamic controller update problem's size grew to 4337 states. The controller that was synthesised in 7.4 s (using a maximum of 46.7 MB) and uploaded, stops the old specification (stopOld) while in region **NoF3**, but the new specification is started (startNew) much later, only when the UAV leaves **NoF3** (see trajectory *Plan_upd*), as it is prohibited in the new specification. The UAV then carries on its new patrol mission of regions **C1** and **D1** as seen in trajectory *Plan_new*.

6.2 Unexpected Battery Consumption Rate

We explore a different scenario of mission plan update: *mission degradation due to unforeseen circumstances*. We assume an original mission that requires covering an area **A** (i.e., visit every discrete location in **A**) for mapping purposes (e.g., [65, 78, 84]).

In large discrete regions, cover missions with explicit-location modelling do not scale well as one fluent for every location is needed to track if it has been covered. Instead, we use an alternative discrete abstraction strategy: Iterator-Based Planning [97]. This follows the idea of sensor-based planning (e.g., [61]) in which synthesised plans rely on sensors implemented in the hybrid control layer. A sensor refers not necessarily to a device and its software that provides information about the physical world but also modules that provide information about the state of the entire software stack below the controller. In this case we use a sensor (is.next.inA?) to identify if a location corresponds to the area to be covered (yes.next.inA) or not (no.next.inA). For such a sensor we will automatically compute the required Python code of its *Hybrid Module* through the *Discretizer* component (i.e., the user may draw on the interactive map a discrete region which translates into a list of locations, which is then embedded into Python code to implement the response to is.next.inA?).

Iterator-Based Planning works by providing a high-level iterator API (see Figure 9a) that allows us to iterate over the discrete locations, abstracting from the number of locations involved. The user can then specify in LTL what must be done for each location as it is iterated over. For example, for a cover **A** mission, every time the iterator responds that there is still a location to process (y.next), the controller should ask is.next.inA?, and it is (yes.next.inA) it should go to that location (see Figure 9b).

The size of the control problem was of 97 states and synthesising a controller for it took 0.5 s (using up to 14.0 MB RAM). We flew two missions, one with the Parrot Ar.Drone and the other with a simulated ArduPlane SITL. For the plane, a region **A** was defined by the user as shown in Figure 8b, discrete cells were of 60 m × 60 m and the flight height of 100 m, generating a universe of 1251 discrete locations. From region **A**, a sensor was automatically computed by the *Discretizer* and uploaded onto the UAV through the *Hybrid Module TX* before starting the original mission plan. The UAV produced the trajectory *Plan_old* shown in Figure 8b while covering **A**.

Suppose that due to wind conditions or a malfunctioning engine, the battery consumption rate is higher than predicted by a runtime monitor as in [91]. At the *Control Panel* level this could fire off an alarm indicating that the UAV will be unable to completely cover region **A**. We simulated this

scenario and had the user intervene (with the UAV still in-flight) by producing a degraded mission plan: the user chooses to reduce to half the original region **A**, to at least have one contiguous region completely covered. This was done first by defining **B** in Figure 8b and then specifying the new mission requirements, together with the standard transition requirement $\Theta_\emptyset$ and the rule (2) that only allows reconfig to happen when *sensor.**A*** is in its initial state. The environment state map $E \not\downarrow_g^{\text{reconfig}} E'$ was defined by ignoring the state of *sensor.**A*** and mapping all states from E to the equivalent in E' with *sensor.**B*** in its initial state.

$$\begin{aligned} \Theta &= \Box(\text{reconfig} \Rightarrow \neg \mathit{SensingA}) \\ \mathit{SensingA} &= \langle \text{is.next.inA?}, \{\text{yes.next.inA}, \text{no.next.inA}\}, \bot \rangle \end{aligned} \tag{2}$$

The *Discretizer* module automatically generates the Python code that implements the new *sensor.**B*** and uploads it onto the flying UAV. Meanwhile the synthesizer produced an update controller in 1.8 s (using up to 16.2 MB and for a size of the dynamic control update problem of 1126 states) that, when uploaded and hotswapped, stopped (stopOld) the old mission, executed reconfig which triggers the binding of *sensor.**B*** module to the *Hybrid Control Layer* (and the unbinding of *sensor.**A***), and signals the start of the new mission (startNew).

The UAV continues its mission (Figure 8b) covering **B** (*Plan_new*).

6.3 Fire Monitoring

UAVs are used to aid firefighters by fire monitoring and tracking [12, 49]. A fire monitoring mission can be as simple as a fire lookout [63] between two locations far apart from each other. Such a mission is suitable for a multi-rotor with stationary flight capabilities. The mission we simulated consists of visiting two areas **A** and **B**, doing a full slow spin (see LTS in Figure 9c) at each of them to have a 360° view of the surrounding area. If the quadcopter has a camera mounted aboard and streams its video to a remote monitoring station, a human can view this footage to detect the presence of fire. From a control problem of 555 states, we synthesised a controller (in 0.3 s using 15.8 MB of RAM) and ran this mission using a simulated ArduCopter SITL and an iterator-based planning approach with discrete cells of 30 m × 30 m and a flight height of 70 m, totalling 1834 discrete locations. The resulting trajectory can be seen as *Plan_old* in Figure 8c.

Our adaptation is useful in this scenario if the human in-the-loop needs a closer look at a certain area where fire is suspected to be present. The user can then select a new region to cover and generate, with help of the *Discretizer*, the required *sensor.**C*** code. To point the camera aboard the UAV downward, the user modifies manually the generated code to move the camera controlling servo when the *Hybrid Module* is initialised. The new mission consists of covering the area **C** and to return to launch when finished.

The environment mapping is similar to the one in Section 6.2: from E (with *sensor.**A*** and *sensor.**B***) to E' (with *sensor.**C***). The transition requirement is a combination of (2) adapted to include *sensor.**B*** and the spin capability in their initial states, Θ_1 and Θ_2 to restrict actions between specifications, and Θ_3 to force a reset of the iterator before startNew (to guarantee full coverage of the region **C**).

$$\begin{aligned} \Theta_1 &= \Box\big((\mathit{OldStopped} \wedge \neg \mathit{NewStarted}) \Rightarrow \neg \mathit{SenseOrMoveCmd}\big) \\ \Theta_2 &= \Box(\text{reconfig} \Rightarrow \mathit{NewStarted}) \\ \Theta_3 &= \Box(\text{startNew} \Rightarrow \mathit{Reset}), \mathit{Reset} = \langle \text{reset}, \text{has.next?}, \bot \rangle \end{aligned} \tag{3}$$

where *SenseOrMoveCmd* is set to true with moving and sensing actions (e.g., takeOff, go, is.next.inA?) and false with the rest.

The update controller was synthesised in 36.8 s (35.1 MB and DCU problem's size of 4047 states) and uploaded immediately. Figure 8c exemplifies well the non-trivial update strategy that the update controller had to execute to satisfy all requirements: the hotSwap occurred while flying to a new location (i.e., just after go.next but before at.next). The controller immediately does stopOld but cannot do startNew as *Reset* does not hold (see Θ_3). Furthermore, to do reset it must not be moving (see Iterator requirements [97]), thus it first waits until the new location is reached (which it can assume will eventually happen), then resets the iterator and then does startNew and reconfig. Between startNew and reconfig, the update controller chooses to do a spin (do.spin) since it does not violate any requirement.

6.4 Unexpected Search & Rescue

Search & rescue scenarios are a common theme in robotics (e.g., [60, 90, 92]) and the flying of the example in Section 1 showcases the ability of our system to adapt from a patrol to a search & rescue mission including uploading of non-trivial functionality.

The original mission is high-height patrol similar to the one in Section 6.1, resulting in a control problem with 159 states which was synthesised in 0.4 s (using up to 16.5 MB) for 48 discrete locations. We flew the synthesised controller using the Parrot Ar.Drone 2.0. The mission is then updated into a low-height flight of the same patrol locations but introducing an image processing *Hybrid Module* to sense at each arrived location for red objects and the following specification, where *Img.processed* is a fluent that is true when the image processing module detects a red object:

$$\forall 0 \leq i, j \leq 47 \cdot \left[i \neq j \wedge \square\big(At.i \rightarrow (\neg At.j \,\mathrm{W}\, Img.processed)\big)\right]$$

This image processing code is uploaded onto the UAV prior to the hotSwap command being issued. The height inconsistency (high- vs low-height) is solved by the following transition specification: $\square\big((OldStopped \wedge \neg NewStarted) \implies \neg(\Gamma \setminus \{\text{low.height}, \text{high.height}\})\big)$, where the set Γ holds all the controllable actions. This dynamic update control problem has a state space of 6102, and synthesis time totalled 15 s (54.6 MB).

7 Discussion, Limitations and Related Work

This paper uses discrete event controller synthesis to ensure correct mission adaptation and as such builds on a large corpus of knowledge developed by the supervisory control [76], reactive synthesis [75] and automatic planning [35] communities. We build in particular on advances in tractability developed by Piterman et. al [74] for GR(1) temporal logic formulas which has polynomial complexity in the size of the control problem.

We build on the notion of models at runtime [15] both in the controllers being enacted, exception mechanisms, and the knowledge repository. We also build on the idea that involving humans in adaptation can add adaptability by having them act as sophisticated sensors and decision-makers or system-level effectors [19, 32, 45]. We believe the possibility of writing high-level mission goals and transition requirements, exploring model based solutions [88] and then deploying them as in this work, facilitates human-in-the-loop adaptation.

Synthesis of discrete events controllers is increasingly being used by the robotics community to produce guaranteed-by-construction reactive plans that command robots [36, 51, 61, 66]. These approaches conceive synthesis mostly at development time. In [45] short-term reachability tasks can be added (and synthesised) at runtime, but the original goal remains fixed. Synthesis has also been used at development time to produce glue code, adapters and mediators (e.g., [6, 44, 50, 73]). Its use at runtime has also been explored to support structural reconfiguration of systems (e.g., [4, 56, 82, 83]).

Runtime change of software systems has been studied extensively. Different application domains and technology stack pose different problems and require different solutions [80]. A major concern is *correctness preservation* throughout change [18]. Many approaches assume that there is no change in the intended system behaviour (i.e., the *specification/mission remains unchanged*) and that a patch is being applied (e.g., [48]), or that this change is small. Alternatively, a set of *fixed domain independent properties*, such as consistency, are expected to hold (e.g., [10, 22, 46, 57]). More recently, complex plans for supporting architectural change while preserving user provided structural constraints has been studied (e.g., [83]). Some approaches do support domain specific specification changes; however, they require a *prespecified universe of possible changes* at the time of running the system for the first time (e.g., [70]).

The need for supporting arbitrary specification changes and *update requirements* that constrain the transition between specifications has been subject of more recent studies (e.g., [47, 77, 96]). Although they focus on *specification and verification of update strategies*, more recently work focuses on *automated synthesis of update plans* (e.g., [11, 56, 69, 71, 86]).

Existing work in this area, however, is *insufficiently expressive* to accommodate the liveness requirements that typical robotic missions have [67]. Furthermore, existing approaches do not address the specifics of temporal mission planning of mobile robots [13, 34] including changes to sensor and actuator abstractions, and changes in discretization. Such changes require reasoning not only about when and how to change the system behaviour but also when to introduce *software reconfiguration* (e.g., binding and unbinding new software components).

The notion of applying discrete event controller synthesis at runtime to support adaptation was proposed in [17]. The MORPH reference architecture proposes a more complex structure in which structural and behavioural adaptation are managed in an independent but coordinated fashion. However, no validation of this concept has been published, as this work does not define concrete mechanisms for ensuring correct adaptation nor does it discuss how to resolve the non-trivial specifics of applying these ideas to a hybrid control architecture [34, 59] needed to address the discrete-continuous gap between mission specifications and the physical world. In a sense, the work presented here provides experimental support to some of the ideas proposed in [17]. In Fig. 6a we present only the components that we implemented from the original MORPH architecture.

Note that our implementation provides a simplified *Controller Manager* implementation that does not support switching between multiple pre-synthesised controllers to allow fast controller update without human-in-the-loop intervention. Although [17] does not prescribe how this may be implemented, existing solutions [29] could be included in our current architecture.

The main simplification introduced with respect to MORPH, and indeed a general limitation of the approach, is the treatment of architectural reconfigurations as atomic. As before, in MORPH a general structure of how complex reconfigurations with multiple steps and uncontrolled outcomes may be managed in coordination with behavioural control of the system under adaptation but no implementation prescriptions are provided. Approaches to synthesis of reconfiguration plans such as [83] could be included into our architecture.

In the context of UAV missions with limited computational resources onboard atomic reconfigurations may be sufficient for a wide variety of adaptations. Reconfigurations may typically involve the introduction, removal or replacement of modules that implement in a lightweight fashion, with few architectural dependencies, specific controllable and uncontrollable events (e.g., a new image processing module, a new manoeuvre for approaching target locations). Our implementation transmits onto the UAV all the modules code in background and reconfiguration is reduced to loading/unloading the required modules into the *Hybrid Control Layer* and a lightweight update of an event/module mapping. Certainly in some adaptive setups with heavyweight components

that must be setup (e.g., servers, databases, etc.) coordination and non-atomicity are critical, but also in some mobile robot adaptation scenarios (notably multi-robot ones) this is also the case.

Machine learning [42] and logic-based learning has been studied as a means to improve adaptivity (e.g., [79]) and also automate the generation of adapted specifications (e.g. [2]). Our work is complementary to these.

Our software architecture is inspired by MORPH [17], the 3 layered architecture for robots [40] and self-managed systems [58], and the MAPE-K architecture [55]. There has been significant work on architectures for robots and adaptive systems (e.g., [3, 20, 37, 39, 41, 83, 89, 95]), see [1] for a systematic study. To the best of our knowledge this is the first one to implement assured adaptation at the mission level without constraining a priori what mission adaptations are allowed.

We believe that the experimental results provide some evidence that runtime synthesis can be used to support mission adaptation in real UAV systems. Of course, there are threats to validity. The main one being that experimental results may not generalise to all UAV setups and vehicle configurations. Certainly, one limitation is that synthesis is being applied to a single UAV and that we are considering only atomic reconfiguration to simplify implementation and presentation (i.e., reconfiguration strategies as in [83] are not supported). Many missions of interest in this domain are multi-vehicle [65], and are currently not handled by the adaptation architecture we propose.

One potential limitation of the approach is that discrete event controller synthesis may not scale to large missions. We restrict the expressiveness of missions so they fit within GR(1) which can be solved polynomially in the size of the control problem. However, the control problem can grow exponentially in states with respect to the number of LTS and formulae. Indeed, synthesis times for the various missions reported in Section 6 differ in up to an order of magnitude (although all under a minute) due to variability in the size of the dynamic update control problem. For missions with many discrete locations, the control problem size can grow combinatorially, but as explained, this can be mitigated using Iterator-Based Planning [97] which has been shown to work for hundreds of thousands of discrete locations. Despite this, scale is an open challenge that needs to be further addressed by the controller synthesis community. A complementary problem to scale is finding an adequate discretization for the workspace taking into account the robot's dynamics, geometry and obstacles [13, 27, 62], aiming to reduce the amount of required discrete locations at the specification level. The results discussed in this paper are orthogonal to any further developments in the field of synthesis, including alternative discretization methods to the grid-based we used in this work.

8 Conclusions

We have presented a novel architecture for UAV systems that supports correct by construction mission adaptation performing synthesis of discrete event controllers at runtime and hotswapping them onto a UAV. The architecture supports both behavioural and structural adaptation building on hybrid control, dynamic controller update and adaptive software architectures. We show in several missions taken from the robotic literature that new mission goals can be introduced and correctly updated into a running system, both for real and simulated scenarios. Having shown how the update problem is non-trivial, we demonstrate how user specified transition requirements can be used to solve inconsistencies and correctly synthesise update strategies, that are guaranteed to take the running system into a state where the software architecture can be reconfigured and the new plan can be executed.

References

[1] A. Ahmad and M. A. Babar. 2016. Software architectures for robotic systems: A systematic mapping study. *Journal of Systems and Software* 122 (2016), 16–39.

[2] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastian Uchitel. 2013. Elaborating Requirements Using Model Checking and Inductive Learning. *IEEE Trans. Software Eng.* 39, 3 (2013), 361–383.

[3] Mehdi Amoui, Mahdi Derakhshanmanesh, Jürgen Ebert, and Ladan Tahvildari. 2012. Achieving dynamic adaptation via management and interpretation of runtime models. *Journal of Systems and Software* 85, 12 (2012), 2720 – 2737. Self-Adaptive Systems.

[4] N. Arshad, D. Heimbigner, and A. L. Wolf. 2007. Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems. *Software Quality Journal* 15, 3 (2007), 265–281.

[5] S. Atoev, K. Kwon, S. Lee, and K. Moon. 2017. Data analysis of the MAVLink communication protocol. In *2017 International Conference on Information Science and Communications Technologies (ICISCT)*. 1–3.

[6] M. Autili, P. Inverardi, F. Mignosi, R. Spalazzese, and M. Tivoli. 2015. Automated Synthesis of Application-Layer Connectors from Automata-Based Specifications. In *Language and Automata Theory and Applications*. Springer International Publishing, Cham, 3–24.

[7] Sabur Baidya, Zoheb Shaikh, and Marco Levorato. 2018. FlyNetSim: An Open Source Synchronized UAV Network Simulator Based on Ns-3 and Ardupilot. In *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems* (Montreal, QC, Canada) *(MSWIM '18)*. ACM, New York, NY, USA, 37–45.

[8] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press.

[9] A. Balkan, M. Vardi, and P. Tabuada. 2018. Mode-Target Games: Reactive Synthesis for Control Applications. *IEEE Trans. Automat. Control* 63, 1 (2018), 196–202.

[10] F. Banno, D. Marletta, G. Pappalardo, and E. Tramontana. 2010. Handling consistent dynamic updates on distributed systems. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*. 471–476.

[11] Luciano Baresi and Carlo Ghezzi. 2010. The Disappearing Boundary Between Development-time and Run-time. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (Santa Fe, New Mexico, USA) *(FoSER '10)*. ACM, New York, NY, USA, 17–22.

[12] E. Beachly, C. Detweiler, S. Elbaum, D. Twidwell, and B. Duncan. 2017. UAS-Rx interface for mission planning, fire tracking, fire ignition, and real-time updating. In *IEEE Intl. Symposium on Safety, Security and Rescue Robotics (SSRR)*. 67–74.

[13] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. J. Pappas. 2007. Symbolic planning and control of robot motion [Grand Challenges of Robotics]. *IEEE Robotics Automation Magazine* 14, 1 (2007), 61–70.

[14] Zachary Birnbaum, Andrey Dolgikh, Victor Skormin, Edward O'Brien, and Daniel Muller. 2014. Unmanned Aerial Vehicle Security Using Recursive Parameter Estimation. In *International Conference on Unmanned Aircraft Systems, ICUAS 2014*, Vol. 84. 692–702.

[15] G. Blair, R. B. France, and N. Bencomo. 2009. Models@ run.time. *Computer* 42, 10 (oct 2009), 22–27.

[16] H. Bouafif, F. Kamoun, F. Iqbal, and A. Marrington. 2018. Drone Forensics: Challenges and New Insights. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 1–6.

[17] Victor Braberman, Nicolas D'Ippolito, Jeff Kramer, Daniel Sykes, and Sebastian Uchitel. 2015. MORPH: A Reference Architecture for Configuration and Behaviour Self-Adaptation. In *Proceedings of the 1st International Workshop on Control Theory for Software Engineering* (Bergamo, Italy) *(CTSE 2015)*. ACM, New York, NY, USA, 9–16.

[18] Radu Calinescu, Danny Weyns, Simos Gerasimou, Muhammad Usman Iftikhar, Ibrahim Habli, and Tim Kelly. 2018. Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1039–1069.

[19] Javier Cámara, Gabriel A. Moreno, and David Garlan. 2015. Reasoning About Human Participation in Self-adaptive Systems. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (Florence, Italy) *(SEAMS '15)*. IEEE Press, Piscataway, NJ, USA, 146–156.

[20] Javier Cámara, Bradley Schmerl, Gabriel A. Moreno, and David Garlan. 2018. MOSAICO: Offline Synthesis of Adaptation Strategy Repertoires with Flexible Trade-Offs. *Automated Software Engg.* 25, 3 (Sept. 2018), 595–626.

[21] A. Chakrabarty, R. Morris, X. Bouyssounouse, and R. Hunt. 2016. Autonomous indoor object tracking with the Parrot AR.Drone. In *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*. 25–30.

[22] Haibo Chen, Jie Yu, Chengqun Hang, Binyu Zang, and Pen-Chung Yew. 2011. Dynamic Software Updating Using a Relaxed Consistency Model. *Software Engineering, IEEE Transactions on* 37, 5 (Sept 2011), 679–694.

[23] H. Choi, M. Geeves, B. Alsalam, and F. Gonzalez. 2016. Open source computer-vision based guidance system for UAVs on-board decision making. In *2016 IEEE Aerospace Conference*. 1–5.

[24] Jean-Françcois Cordeau, Gilbert Laporte, and Stefan Ropke. 2008. *Recent Models and Algorithms for One-to-One Pickup and Delivery Problems*. Springer US, Boston, MA, 327–357.

[25] J. d. S. Barros, T. Oliveira, V. Nigam, and A. V. Brito. 2016. A Framework for the Analysis of UAV Strategies Using Co-simulation. In *2016 VI Brazilian Symposium on Computing Systems Engineering (SBESC)*. 9–15.

[26] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. Springer, 1–32.

[27] J. A. DeCastro, V. Raman, and H. Kress-Gazit. 2015. Dynamics-driven adaptive abstraction for reactive high-level mission and motion planning. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 369–376.

[28] Ahmet Denker and Mehmet Can Iseri. 2017. Design and implementation of a semi-autonomous mobile search and rescue robot: SALVOR. In *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*. 1–6.

[29] Nicolás D'Ippolito, Víctor Braberman, Jeff Kramer, Jeff Magee, Daniel Sykes, and Sebastian Uchitel. 2014. Hope for the Best, Prepare for the Worst: Multi-tier Control for Adaptive Systems. In *Proc. of the 36th Int. Conf. on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 688–699.

[30] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. 2011. Synthesis of live behaviour models for fallible domains. In *2011 33rd International Conference on Software Engineering (ICSE)*. 211–220.

[31] N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. 2008. MTSA: The Modal Transition System Analyser. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 475–476.

[32] Christoph Dorn and Richard N. Taylor. 2013. Coupling Software Architecture and Human Architecture for Collaboration-aware System Adaptation. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 53–62.

[33] S. Dubal, M. Yadav, V. Singh, V. Uniyal, and M. Singh. 2016. Smart Aero-Amphibian Surveillance System. In *International Conference Workshop on Electronics Telecommunication Engineering (ICWET 2016)*. 112–116.

[34] G. E. Fainekos, H. Kress-Gazit, and G. J. Pappas. 2005. Temporal Logic Motion Planning for Mobile Robots. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. 2020–2025.

[35] Richard E. Fikes and Nils J. Nilsson. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI'71)*. 608–620.

[36] C. Finucane, Gangyuan Jing, and H. Kress-Gazit. 2010. LTLMoP: Experimenting with language, Temporal Logic and robot control. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 1988–1993.

[37] S. García, C. Menghi, P. Pelliccione, T. Berger, and R. Wohlrab. 2018. An Architecture for Decentralized, Collaborative, and Autonomous Robots. In *2018 IEEE International Conference on Software Architecture (ICSA)*. 75–7509.

[38] Sergio García, Patrizio Pelliccione, Claudio Menghi, Thorsten Berger, and Tomas Bures. 2019. High-level Mission Specification for Multiple Robots. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering* (Athens, Greece) *(SLE 2019)*. ACM, New York, NY, USA, 127–140.

[39] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer* 37, 10 (Oct. 2004), 46–54.

[40] E. Gat. 1997. *Three-layer Architectures, Artificial Intelligence and Mobile Robots*. MIT/AAAI Press.

[41] I. Gerostathopoulos, T. Bures, P. Hnetynka, A. Hujecek, and D. Plasil, F.and Skoda. 2017. Strengthening Adaptation in Cyber-Physical Systems via Meta-Adaptation Strategies. *ACM Transactions on Cyber-Physical Systems* 1 (04 2017), 1–25.

[42] Omid Gheibi, Danny Weyns, and Federico Quin. 2021. Applying Machine Learning in Self-Adaptive Systems: A Systematic Literature Review. *ACM Trans. Auton. Adapt. Syst.* 15, 3, Article 9 (Aug. 2021), 37 pages.

[43] Dimitra Giannakopoulou and Jeff Magee. 2003. Fluent Model Checking for Event-based Systems. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Helsinki, Finland) *(ESEC/FSE-11)*. ACM, New York, NY, USA, 257–266.

[44] Joel Greenyer, Christian Brenner, Maxime Cordy, Patrick Heymans, and Erika Gressi. 2013. Incrementally Synthesizing Controllers from Scenario-based Product Line Specifications. In *Proc. of the 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. New York, NY, USA, 433–443.

[45] M. Guo, S. Andersson, and D. V. Dimarogonas. 2018. Human-in-the-Loop Mixed-Initiative Control Under Temporal Tasks. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 6395–6400.

[46] Deepak Gupta, Pankaj Jalote, and Gautam Barua. 1996. A Formal Framework for On-line Software Version Change. *IEEE Trans. Software Eng.* 22, 2 (1996), 120–131.

[47] Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. 2012. Specifying and Verifying the Correctness of Dynamic Software Updates. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments* (Philadelphia, PA) *(VSTTE'12)*. Springer-Verlag, Berlin, Heidelberg, 278–293.

[48] Petr Hosek and Cristian Cadar. 2013. Safe Software Updates via Multi-version Execution. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 612–621.

[49] Lizhong Hua and Guofan Shao. 2016. The progress of operational forest fire monitoring with infrared remote sensing. *Journal of Forestry Research* 28 (12 2016), 1–15.

[50] Valérie Issarny, Amel Bennaceur, and Yérom-David Bromberg. 2011. *Middleware-Layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability.* Springer Berlin Heidelberg, Berlin, Heidelberg, 217–255.
[51] Gangyuan Jing, Tarik Tosun, Mark Yim, and Hadas Kress-Gazit. 2017. An End-to-End System for Accomplishing Tasks with Modular Robots: Perspectives for the AI community. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17.* 4879–4883.
[52] Marcin Jurdzinski. 2000. Small Progress Measures for Solving Parity Games. In *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science (STACS '00).* Springer-Verlag, Berlin, Heidelberg, 290–301.
[53] Sudeep Juvekar and Nir Piterman. 2006. Minimizing Generalized Büchi Automata. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 45–58.
[54] Robert Keller. 1976. Formal Verification of Parallel Programs. *Commun. ACM* 19 (07 1976), 371–384.
[55] J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (Jan 2003), 41–50.
[56] Narges Khakpour, Farhad Arbab, and Eric Rutten. 2015. Synthesizing structural and behavioral control for reconfigurations in component-based systems. *Formal Aspects of Computing* (Dec. 2015).
[57] Jeff Kramer and Jeff Magee. 1990. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Softw. Eng.* 16, 11 (Nov. 1990), 1293–1306.
[58] Jeff Kramer and Jeff Magee. 2007. Self-Managed Systems: an Architectural Challenge. In *2007 Future of Software Engineering (FOSE '07).* IEEE Computer Society, Washington, DC, USA, 259–268.
[59] H. Kress-Gazit, G. Fainekos, and G. Pappas. 2008. Translating Structured English to Robot Controllers. *Advanced Robotics* 22 (10 2008), 1343–1359.
[60] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. 2007. Where's Waldo? Sensor-Based Temporal Logic Motion Planning. In *Proceedings 2007 IEEE International Conference on Robotics and Automation.* 3116–3121.
[61] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. 2009. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE Transactions on Robotics* 25, 6 (2009), 1370–1381.
[62] Hadas Kress-Gazit, Morteza Lahijanian, and Vasumathi Raman. 2018. Synthesis for Robots: Guarantees and Feedback for Robot Behavior. *Annual Review of Control, Robotics, and Autonomous Systems* 1, 1 (2018), 211–236.
[63] Omer Kucuk, Ozer Topaloglu, Arif Altunel, and Mehmet Cetin. 2017. Visibility analysis of fire lookout towers in the Boyabat State Forest Enterprise in Turkey. *Environmental Monitoring and Assessment* 189 (06 2017).
[64] S. C. Livingston and R. M. Murray. 2013. Just-in-time synthesis for reactive motion planning with temporal logic. In *2013 IEEE International Conference on Robotics and Automation.* 5048–5053.
[65] Tauã M. Cabreira, Lisane Brisolara, and Paulo Ferreira Jr. 2019. Survey on Coverage Path Planning with Unmanned Aerial Vehicles. *Drones* 3 (01 2019), 4.
[66] S. Maniatopoulos, P. Schillinger, V. Pong, D. C. Conner, and H. Kress-Gazit. 2016. Reactive high-level behavior synthesis for an Atlas humanoid robot. In *2016 IEEE International Conference on Robotics and Automation (ICRA).* 4192–4199.
[67] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger. 2019. Specification Patterns for Robotic Missions. *IEEE Transactions on Software Engineering* (2019), 1–1.
[68] Mark O. Milhouse. 2015. Framework for Autonomous Delivery Drones. In *Proceedings of the 4th Annual ACM Conference on Research in Information Technology.* 1–4.
[69] L. Nahabedian, V. Braberman, N. D'Ippolito, S. Honiden, J. Kramer, K. Tei, and S. Uchitel. 2018. Dynamic Update of Discrete Event Controllers. *IEEE Transactions on Software Engineering* (2018), 1–1.
[70] A. Nooruldeen and K. W. Schmidt. 2015. State Attraction Under Language Specification for the Reconfiguration of Discrete Event Systems. *IEEE Trans. on Automatic Control* 60, 6 (June 2015), 1630–1634.
[71] V. Panzica La Manna, J. Greenyer, C. Ghezzi, and C. Brenner. 2013. 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, 63–72.
[72] E. Pecker-Marcosig, S. Zudaire, M. Garrett, S. Uchitel, and R. Castro. 2020. Unified DEVS-based platform for Modeling and Simulation of Hybrid Control Systems. In *2020 Winter Simulation Conference.* 1051–1062.
[73] Patrizio Pelliccione, Massimo Tivoli, Antonio Bucchiarone, and Andrea Polini. 2008. An Architectural Approach to the Correct and Automatic Assembly of Evolving Component-based Systems. *J. Syst. Softw.* 81, 12 (Dec. 2008), 2237–2251.
[74] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2006. Synthesis of reactive (1) designs. *Lecture notes in computer science* 3855 (2006), 364–380.
[75] A. Pnueli and R. Rosner. 1989. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Austin, Texas, United States) *(POPL '89).* ACM, New York, NY, USA, 179–190.
[76] Peter JG Ramadge and W Murray Wonham. 1989. The control of discrete event systems. *Proc. of the IEEE* 77, 1 (1989), 81–98.
[77] Andres J. Ramirez, Betty H.C. Cheng, Philip K. McKinley, and Benjamin E. Beckmann. 2010. Automatically Generating Adaptive Logic to Balance Non-functional Tradeoffs During Reconfiguration. In *Proc. of the 7th Int. Conf. on Autonomic*

Computing (Washington, DC, USA) *(ICAC '10)*. ACM, New York, NY, USA, 225–234.

[78] C. A. Rokhmana and R. Andaru. 2016. Utilizing UAV-based mapping in post disaster volcano eruption. In *2016 6th International Annual Engineering Seminar (InAES)*. 202–205.

[79] Stefan Rudolph, Sven Tomforde, and Jörg Hähner. 2019. Mutual Influence-Aware Runtime Learning of Self-Adaptation Behavior. *ACM Trans. Auton. Adapt. Syst.* 14, 1, Article 4 (Sept. 2019), 37 pages.

[80] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. 2013. A survey of dynamic software updating. *Journal of Software: Evolution and Process* 25, 5 (2013), 535–568.

[81] A. Sinisterra, M. Dhanak, and N. Kouvaras. 2017. A USV platform for surface autonomy. In *OCEANS 2017 - Anchorage*. 1–8.

[82] D. Sykes, W. Heaven, J. Magee, and J. Kramer. 2008. From Goals to Components: A Combined Approach to Self-Management. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08)*.

[83] Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic. 2010. PLASMA: A Plan-Based Layered Architecture for Software Model-Driven Adaptation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. ACM, New York, NY, USA, 467–476.

[84] A. Tariq, S. M. Osama, and A. Gillani. 2016. Development of a Low Cost and Light Weight UAV for Photogrammetry and Precision Land Mapping Using Aerial Imagery. In *2016 International Conference on Frontiers of Information Technology (FIT)*. 360–364.

[85] CanberraUAV OBC team. 2020. MAVProxy Software Package. Retrieved January 16, 2020 from https://github.com/ArduPilot/MAVProxy

[86] Sander Thuijsman and Michel Reniers. 2020. Transformational Supervisor Synthesis for Evolving Systems. *IFAC-PapersOnLine* 53, 4 (2020), 309–316. 15th IFAC Workshop on Discrete Event Systems WODES 2020 - Rio de Janeiro, Brazil, 11-13 November 2020.

[87] Paolo Toth, Daniele Vigo, Paolo Toth, and Daniele Vigo. 2014. *Vehicle Routing: Problems, Methods, and Applications, Second Edition.* Society for Industrial and Applied Mathematics, USA.

[88] Sebastián Uchitel, Robert Chatley, Jeff Kramer, and Jeff Magee. 2004. Fluent-Based Animation: Exploiting the Relation between Goals and Scenarios for Requirements Validation. In *RE*. 208–217.

[89] Thomas Vogel and Holger Giese. 2014. Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Trans. Auton. Adapt. Syst.* 8, 4, Article 18 (Jan. 2014), 33 pages.

[90] J. Wang, W. Chen, and V. Temu. 2018. Multi-Vehicle Motion Planning for Search and Tracking. In *2018 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*. 352–355.

[91] M. Wei and V. Isler. 2018. Coverage Path Planning Under the Energy Constraint. In *2018 IEEE International Conference on Robotics and Automation*. 368–373.

[92] Y. Yamazaki, M. Tamaki, C. Premachandra, C. J. Perera, S. Sumathipala, and B. H. Sudantha. 2019. Victim Detection Using UAV with On-board Voice Recognition System. In *IEEE International Conference on Robotic Computing*. 555–559.

[93] Z. Yan, J. He, and J. Li. 2017. An improved multi-AUV patrol path planning method. In *2017 IEEE International Conference on Mechatronics and Automation (ICMA)*. 1930–1936.

[94] Y. Yang, J. Zhao, X. Yin, and S. Li. 2019. Formal Synthesis of Warehouse Robotic Systems with Temporal Logic Specifications. In *2019 IEEE International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*. 469–474.

[95] Edith Zavala, Xavier Franch, and Jordi Marco. 2019. HAFLoop: An architecture for supporting Highly Adaptive Feedback Loops in self-adaptive systems. *Future Generation Computer Systems* 105 (12 2019).

[96] Ji Zhang and Betty HC Cheng. 2006. Model-based development of dynamically adaptive software. In *Proc. of the 28th Int. Conf. on Software engineering*. ACM, 371–380.

[97] S. Zudaire, M. Garrett, and S. Uchitel. 2020. Iterator-Based Temporal Logic Task Planning. In *2020 International Conference on Robotics and Automation (ICRA)*.

[98] S. Zudaire, L. Nahabedian, and S. Uchitel. 2021. Fork of MAVProxy to support hybrid controllers and assured mission adaptation. Retrieved October 12, 2021 from https://bitbucket.org/szudaire/modules

[99] S. Zudaire, L. Nahabedian, and S. Uchitel. 2021. MTSA tool, LTS requirements and mission videos. Retrieved October 12, 2021 from http://mtsa.dc.uba.ar/uav/uav-morph.html