Abstract—
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 propose applying the MORPH adaptive software architecture to UAVs and show how controller synthesis can be used both to guarantee correct transitioning from the old to the new mission goals while 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.

I. INTRODUCTION

Adaptive systems are capable of changing their behaviour while running in response to changes in their environment, capabilities and goals [1]. 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 goals that the robot must achieve.

Consider an Unmanned Aerial Vehicle (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, 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 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.

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.

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. The plan is executed on a hybrid control architecture that supports runtime swapping of plans, and runtime binding and unbinding of hybrid components.

A. Related Work

Runtime change of software systems has been studied extensively. Different application domains and technology stacks pose different problems and require different solutions [2]. A major concern is correctness preservation throughout change. 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., [3]). Alternatively, a set of fixed domain independent properties, such as consistency, are expected to hold (e.g., [4], [5], [6], [7]). More recently, complex plans for supporting architectural change while preserving user provided structural constraints has been studied (e.g., [8]). 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., [9]).

The need for supporting arbitrary specification changes and update requirements that constrain the transition between spec-
ifications has been subject of more recent studies (e.g., [10], [11], [12]). Although they focus on specification and verification of update strategies, more recently work focuses on automated synthesis of update plans (e.g., [13], [14], [15]).

Existing work in this area, however, is insufficiently expressive to accommodate the liveness requirements that typical robotic missions have [16]. Furthermore, existing approaches do not address the specifics of temporal mission planning of mobile robots [17], [18] including changes to sensor and actuator abstractions, including 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). A proposal for the latter has been outlined in the MORPH reference architecture [19] but does not define mechanisms for ensuring correct adaptation nor does it resolve the non-trivial specifics of applying these ideas to a hybrid control architecture [18], [20] required to resolve the discrete-continuous gap between mission specifications and the physical world.

B. Summary

The main contributions of this paper is a system for adapting UAV missions with correctness guarantees that a) builds on discrete event controller update [15] but extends it to liveness properties to support typical mobile robot missions [16], b) that implements a hybrid controller [20] architecture that incorporates reconfiguration capabilities from [19]. 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.

In Section III we show how UAV missions and mission adaptations can be specified. This sets the appropriate level of abstraction to present an overview of the architecture (Section IV-A) and detailed description of the software architecture and main software components when applied to UAVs (Section IV-B). We then report on our validation efforts in Section V and conclude with a discussion and future work (Section VI).

II. PRELIMINARIES

A. Labelled Transition Systems (LTS)

The dynamics of the interaction of the robot with its environment are modelled using LTS [21], which are automata where transitions are labelled with events that constitute the interactions of the modelled system with its environment. We partition events into controlled and uncontrolled to specify assumptions about the environment and safety requirements for a controller. 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. We use an interrupt operator [15] \( E'^{\ell} \) to model that the behaviour described by LTS \( E \) may be interrupted by event \( \ell \) to become LTS \( E' \). Function \( f \) sets the initial state of \( E' \) based on the state of \( E \) when the interrupt happens.

B. Fluent Linear Temporal Logic (FLTL)

In order to describe environment assumptions and system goals it is common to use formal languages like FLTL [22], a variant of linear-time temporal logic that uses fluents to describe states over sequences of actions. A fluent \( \beta = (Set_+, Set_-, v) \) is defined by a set of initiating actions \( Set_+ \), a set of terminating actions \( Set_- \), and an initial value \( v \) true (\( \top \)) or false (\( \bot \)). We may omit set notation for singletons and use an action label \( \ell \) for the fluent defined as \( \beta = (\ell, Act \{ \ell \}, \bot) \). Thus, the fluent \( \ell \) is only true just after the occurrence of the action \( \ell \). FLTL is defined similarly to propositional LTL but where a fluent holds at a position \( i \) in a trace \( \pi \) based on the events occurring in \( \pi \) up to \( i \). Temporal connectives are interpreted as usual: \( \odot \varphi \), \( \Box \varphi \), and \( \varphi W \psi \) mean that \( \varphi \) eventually holds, always holds, and (weakly) holds until \( \psi \), respectively. An LTS \( E \) satisfies \( \varphi \ (E \models \varphi) \) when all its traces satisfy \( \varphi \). We refer to liveness formulae as those that only have infinite trace violations. Otherwise we refer to them as safety formulae.

C. Discrete Event Controller Synthesis

We adopt the controller synthesis formulation from [23]. Given an LTS \( E \) describing the execution environment of a discrete controller with a set of controllable actions \( L \) and a task specification \( \varphi \) expressed in FLTL, the goal of controller synthesis is to find an LTS \( C \) such that \( E||C : (1) \) is deadlock free, (2) \( C \) does not block any non-controlled actions, and (3) \( E[C \models \varphi] \) We say that a control program \( (E, \varphi, L) \) is realizable if 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 \( \varphi \)) and also on the fragment of the logic used for \( \varphi \). When goals are restricted to safety formulae and GR(1) formulae the control problem can be solved in polynomial time [24]. GR(1) formulas are of the form \( \bigwedge_{i=1}^n \Box \varphi \) \( A_i \implies \bigwedge_{i=1}^m \Box \varphi \) \( G_i \) where \( A_i \) and \( G_i \) are Boolean combinations of fluents. In this paper we use MTSAS [25] for solving control problems.

D. Dynamic Controller Update

We summarise the results in [15]. Assume a controller \( C \) that is solution to a control problem \( (E, \varphi, L) \) that is to be replaced by a new controller \( C' \), with \( \varphi = \Box G \) and \( G \) a boolean combination of fluents. The term \( (C[G(\text{hotSwap} \ C')]_f \) models hot-swapping one controller with the other, where \( C' \) is initialised based on the current state of \( C \) at the time of hotSwap using function \( f \). Event hotSwap is uncontrolled.

Assume the intention is that the new controller \( C' \) guarantees \( \varphi' \) in a new execution environment \( E' \), with \( \varphi' = \Box G' \) and \( G' \) a boolean combination of fluents. In addition, consider a function \( g \) that initialises the state of \( E' \) based on the current state of \( E \) and a controlled event \( \text{reconfig} \) that models the reconfiguration of the execution environment \( E \) by \( E' \), i.e., \( E'[\text{reconfig} \ E'] \).

In addition, we introduce two more controlled events: stopOld that signals from when \( \varphi \) is no longer guaranteed (i.e., \( G \ W \text{stopOld} \)) and startNew that signals from when \( \varphi' \) is guaranteed (i.e., \( \Box (\text{startNew} \implies \varphi') \)).
Finally, assume a safety FLTL formula Θ that models the transition requirement between controllers. In other words, constrains the occurrence of reconfig, startNew and stopOld. As an example consider a standard domain independent requirement \( \Theta_0 = \Box (\neg \text{OldStopped} \lor \text{NewStarted}) \), with fluents OldStopped and NewStarted turning on with the occurrence of stopOld and startNew, respectively, and never turning off. \( \Theta_0 \) states that the system must always be under control to achieve the old specification (φ) or the new one (φ').

The dynamic controller update problem is to find \( C' \) and \( f \) such that (1) \( f \) is a total function, (2) \( C|_{t_f}^{\text{hotswap}}C' \) does not block any non-controlled actions in \( E|_{t_f}^{\text{reconfig}}E' \), and that \( C|_{t_f}^{\text{hotswap}}C'||E|_{t_f}^{\text{reconfig}}E' \) is (3) deadlock free, and (4) satisfies \( \varphi_{DCU} \) defined as the conjunction of the following:

1) \( G \ W \ \text{stopOld} \)
2) \( \Theta \)
3) \( \Box (\text{startNew} \implies \phi') \)
4) \( \Box (\text{hotswap} \implies (\Box \text{oldSpec} \land \Box \text{reconfig} \land \Box \text{startNew})) \)

Note that conjunct (4) of \( \varphi_{DCU} \) requires that once the uncontrolled event hotswap occurs, the system must eventually reconfigure and switch specifications.

In [15], DCU problem is reduced to a standard discrete event control problem, and tool (MTSA [25]) is reported that computes an LTS \( C_0 \) of the form for \( C|_{t_f}^{\text{hotswap}}C' \) from which \( C' \) and \( f \) can be extracted.

E. Dynamic Controller Update of Live Missions

The DCU problem in [15] is restricted to safety requirements. However, a typical mobile robot mission will have liveness properties. For instance, and taking patterns recollected from scientific literature and industrial case studies [16], patrolling two locations requires \( \Box \Diamond ' \text{at1} \land \Box \Diamond ' \text{at2} \) and a delivery mission requires \( \Box (\text{available} \implies \Diamond \text{deliver}) \).

Simply lifting the safety restriction over φ and φ' leads to problems. Consider \( \varphi = \Box (\text{available} \implies \Diamond \text{deliver}) \) and the requirement that the update must satisfy: \( \Box (\text{available} \implies \Diamond \text{deliver}) \) W \( \text{stopOld} \). Such a requirement effectively does not allow scenarios in which it is necessary to abort a pending delivery obligations.

To provide a more general framework for updating controllers, we redefine the property \( \varphi_{DCU} \) that \( C|_{t_f}^{\text{hotswap}}C'||E|_{t_f}^{\text{reconfig}}E' \) is expected to satisfy. We assume without loss of generality that a mission specifications \( \varphi \) is split into a safety part (φS) and a liveness part φL. We define \( \varphi_{DCU} \) as:

1) \( \varphi_S \ W \ \text{stopOld} \)
2) \( \Theta \)
3) \( \Box (\text{startNew} \implies \phi') \)
4) \( \Box (\text{hotswap} \implies (\Box \text{oldSpec} \land \Box \text{reconfig} \land \Box \text{startNew})) \)

Note that in (1) there is no mention of the liveness part of the old specification (φL), consequently all liveness obligations will be dropped as soon as the new controller is put in place. Should this not be desired, Θ can be defined to prevent this (e.g., \( \Theta = (\Box (\text{available} \implies \Diamond \text{deliver}) \) W \( \text{stopOld} \)).

If \( \phi' = \varphi_S \land \varphi_L \) and \( \varphi_L \) is of the form \( \bigwedge_{i=1}^{m} \Box \Diamond G_i \), then items (3) and (4) can be combined to conform to an equivalent safety plus GR(1) formula: \( \varphi'_S \land \Box \Diamond \text{HotSwap} \implies \Box (\text{OldStopped} \land \text{NewStarted} \land \text{Reconfigured}) \land \phi'_L \), where \( \text{OldStopped}, \text{NewStarted}, \) and \( \text{Reconfigured} \) are fluents initially false that become true once stopOld, startNew and reconfig occur. Thus, a similar pattern as described in [15] can be applied to polynomially solve DCU problems with recurrent liveness missions (i.e., \( \bigwedge_{i=1}^{m} \Box \Diamond G_i \)). We have extended the implementation of the MTSA [25] tool to support solving these control problems.

III. 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 solved using DCU. We illustrate the importance of obtaining one controller that works for any reachable state of the current mission plan (i.e. \( C|_{t_f}^{\text{hotswap}}C' \)). Nonetheless, this example is simple in that the term \( E|_{t_f}^{\text{reconfig}}E' \) of the DCU control problem allows \( E = E' \) and \( g \) be the identity function. The transition requirement \( \Theta \) 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 \( \Theta \) to solve adaptation problems in which the new and old missions are logically inconsistent.

A. Adaptation of Live Missions

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. In Fig.1 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. [15] 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. [16]. Note that the location of the UAV is marked as unknown in Fig. [16]. As the UAV is constantly moving while the 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. [20] we show a portion of the LTS covering only cells \( 0 - 2, 4 - 6 \), where we model the movement actions as control modes [26].
with a controllable (go.i) and uncontrollable (at.i) pair. We also model the grab and release mechanisms for the three package types p1, p2, p3 as controllable actions using the LTS in Fig. 2a. Note that the initial location of the UAV is modelled by the initial state of the LTS of Fig. 2a.

We now formalise the mission goals, specifying where each package type can be grabbed and released. For instance, we define a safety property $\varphi_{p1} = \Box((\text{grab.1} \implies \text{At.4}) \land (\text{release.1} \implies \text{At.3}))$ to require packages of type $p1$ be taken from A to C. Fluents $\text{At.i} = \{(\text{at}i)\}$, $\{(\text{go.k} \leq k \leq 11)\}$ are true when the robot is at location $i$. Additionally, we require $\psi_{p1} = \Box(\text{Carrying.i} \land \text{At.3} \implies (\lnot \text{Moving W release.1}))$ to ensure that the UAV will deposit packets as soon as it arrives at the respective target location, with fluents $\text{Moving}$ and $\text{Carrying.i}$ being turned on/off with go/at actions and when the UAV does a grab/release of package $p1$, respectively. Avoiding empty trips is accomplished adding another safety specification: $\gamma = \Box(\text{Moving} \implies \text{Carrying.i} \lor \text{Carrying.2} \lor \text{Carrying.3})$.

Finally, we add the liveness property of continuously delivering packages $p1, p2, p3$: $\rho = \Box(\text{release.1} \land \Box(\text{release.2} \land \Box(\text{release.3}$. We refer to $\rho \land \gamma \land \bigwedge_{i=1,\ldots,3} \varphi_{p1} \land \psi_{p1}$ as OLDSPEC.

A controller $C$ for this mission can be automatically built (as discussed in Sec. II-C) 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 MTSAs) exhibits the following trace: $\text{grab.1, go.8, at.8, go.9, at.9, go.10, at.10, grab.3, go.11, at.11, go.7, at.7, go.3, at.3, release.3, release.1, grab.2, go.2, \ldots}$. 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 $\varphi_{p1}$ and $\psi_{p1}$ must be changed slightly to reflect the new delivery relations shown in Fig. 1c. Assume these properties to be $\varphi'_{p1}$ and $\psi'_{p1}$. We refer to $\rho \land \gamma \land \bigwedge_{i=1,\ldots,3} \varphi'_{p1} \land \psi'_{p1}$ 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 $\Theta$ 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

---

Fig. 1: Package delivery. Plans in (b), (d-f) are 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). We omitted the occurrence of the event reconfig since in scenarios (d-f) reconfiguration is trivial.

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

---
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 $\Theta_0 = \Box(\neg OldStopped \lor NewStarted)$ mentioned in Section II-D 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 $\text{NewSpec}$, $E'$, $g$, and $\Theta$, and given that the current mobile robot is running a controller $C$ to achieve $\text{OldSpec}$ in environment $E$, we have a fully formulated DCU problem (with live missions, see Section II-E) for which a solution $(C$ and $f)$ can be constructed.

Consider the scenarios in Fig. 1d and 1e in which the UAV controlled by $C$ is flying towards location C carrying two packages as part of its plan to satisfy $\text{OldSpec}$ 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. 1d while $C'$ is being computed the UAV reaches location C and, as per $\text{OldSpec}$ 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 $\text{NewSpec}$ will hold from now on (startNew) and that $\text{OldSpec}$ will not be guaranteed anymore (stopOld). It does so in this order to comply to $\Theta$. 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 $\text{NewSpec}$.

In Fig. 1e $C'$ 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 $p_4$ 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 $\text{NewSpec}$. Instead, it picks up $p_2$ and continues pickups and drop offs as per $\text{NewSpec}$.

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 howswapping time 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 $\text{NewSpec}$ an additional requirement forbidding transportation of three packages (to avoid overstraining the UAV): $\Box(\neg (\text{Carrying.1} \lor \text{Carrying.2} \lor \text{Carrying.3})$. Assume that similarly to Fig. 1e the computation of $C'$ and $f$ terminates before the UAV reaches location C (see Fig. 11). Here, the new controller cannot immediately start satisfying $\text{NewSpec}$ as picking up $p_2$ would violate the requirements. Hence, it chooses to delay the change of specification, first dropping off $p_1$ and $p_3$, and also picking up $p_2$ as required in $\text{OldSpec}$. 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.

### B. 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 o 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 changes in the set of controllable and non-controllable events, the formulation of the environment LTS $E'$ and mission goals $\varphi'$.

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 current discretized workspace. In addition, a new package type $p_4$ is required to be transported. The delivery requirements are shown in Fig. 3a 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. 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 3b). The second, is the reconfiguration of the workspace discretization, for which we use the simplified workspace change depicted in Fig. 3a. 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. 3d. Note that reconfiguration can only 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.

Composing the LTS in Fig. 3b for $p_4$, with the models for $p_1, \ldots, p_3$ (see Fig. 2b) and one along the lines of Fig. 3a generates a model for $E_{k}^{\text{reconfig}}E'$ (instead of providing $E'$ and $g$ separately).

The new delivery requirements are modelled according to Fig. 3a must be included in $\text{NewSpec}$, and a transition
two missions are logically inconsistent, there is no safe state
which we used in the previous scenarios is not adequate: If the
transition requirement

\( C \in \Theta \)

requirement \( \Theta \) must be provided. We assume the simple \( \Theta_0 \)
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. 4 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 \( \text{reconfig} \), 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, ..., −4). At −3, the new module is used to pick
up a \( p_4 \) package. Control proceeds satisfying the new mission
requirements.

C. 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 \( \Theta_0 = \Box (\neg \text{OldStopped} \lor \text{NewStarted}) \)
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. We illustrate this
in a simple example.

Example 3 (Surveillance Update): Consider a typical UAV
patrol mission as described in [16] for surveillance of two
areas \( A_1 \) and \( A_2 \) as shown in Fig. 4a. 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. 4b.

The original UAV mission goal can be written as:

\( (\Box \neg \text{At.0} \land \Box \neg \text{At.4}) \land (\Box \neg \neg \text{At.0}, \text{NoFLYOLD}) \).

That is, be at cells 0 and 4 infinitely often and never be in the
NoFLY-
OLD region. Similarly, the new mission can be specified as

\( (\Box \neg \text{At.3} \land \Box \neg \text{At.5}) \land (\Box \neg \neg \text{At.0}, \text{NoFLYNEW}) \),

using appropriately
defined fluents.

Note that in this example if we used the transition re-
requirement \( \Theta_0 \) the DCU problem has no solution, as is not
possible 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 inconsis-
tencies 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 mis-
ion requirements perform arbitrary actions before performing
startNew. Note that the latter must eventually occur as \( \Diamond \text{startNew} \)
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 is dropped, the UAV must be at
locations 4 or 5 until the new one is adopted:

$$\Theta = \Box \left( \text{OldStopped} \implies ((\text{At.4} \lor \text{At.5}) \land \text{NewStarted}) \right)$$ (1)

With this transition requirement, it is possible to synthesise a mission to satisfy the mission adaptation.

IV. 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 [27] 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 [19] reference architecture and integrating them with hybrid controllers. MORPH (Figure 5a) 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 5b). Secondly, we provide implementation details of the architecture that supports assured mission adaptation of UAVs.

A. Architectural Extensions for Adaptability

Hybrid controllers serve as an interface between the discrete high-level events of the synthesised controllers and the low-level sensors, feedback-controllers and other actuators from a robot. As a result, these architectures help produce continuous movement and trajectories that satisfy the user specification. In terms of MORPH, this hybrid layer lives in the **Target System** , while the continuous execution of the discrete event controller is what occurs within the **Controller Enactor** component (see Figure 5a). The MORPH **Target System** is implemented in our architecture by the **Robot Layer** and the **Hybrid Control Layer** (see Figure 5b). The MORPH **Effectors** map to **Actuators** and **Actuators**, while **Probes** map to **Sensors**, together with the required **Hybrid Modules** to make them work.

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, $C^{\text{hotSwap}}_f$ 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.

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. 5b) 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 5a 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 request the **Module Loader** loads and unloads the corresponding modules into the **Hybrid Control Layer**.

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 select discrete regions and their granularity to aid in the specification of the synthesis problems, automatically generating
representations in Python code, LTS and FLTL that are used by the Model Editor and Code Editor components.

- **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.

**B. 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.

The system (see Fig. 5b) 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 [29], 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 [29], [30], 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 [31], [32]. The SITL simulator allows us to test UAV systems loaded with ArduPilot firmware (e.g., custom made as in [33] or commercial as the 3DR Solo Drone [31]) without the UAV hardware. This simulator (see [34]) has been used to seamlessly go from testing to actually flying a custom made fixed-wing vehicle (e.g., [35]) based on a Pixhawk (e.g., [36], [37]).

Similarly to [38], [39], [40] 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 [25]. 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 [41]. This is a widely used (e.g., [37], [38], [41], [42]) 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., [37]). 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., [38]) 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 [II] and provides a back-end *Synthesiser* layer.

**V. 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:

- *Feasability* by running multiple missions and informally validating that the resulting UAV behaviour is consistent with the intended behaviour. This includes analysing if synthesis times and memory consumption are a problem for realistic missions.

- *UAV flexibility* by using different UAVs. We flew a Parrot Ar.Drone in Section [V A] and [V D] and ArduPilot simulations for fixed-wing UAVs (ArduPlane) and quadcopters (ArduCopter) in Sections [V B] and [V C] respectively.

- *Hybrid Control Layer flexibility* by implementing two different abstraction approaches: Iterator-Based Planning (Sections [V B] and [V C]) and Explicit-Location Planning (Sections [V A] and [V D]).

- *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 [16] 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 supplementary material.

**A. Unexpected Goal Change**

We revisit the Example 3 of Section [III-C]. The original mission consists of a typical patrol mission as in [16], [33], [43], [44], [45], [46] 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 6a. For this mission we used discrete cells of $10 \times 10$ 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 [III-A] a safety rule $\square \neg \text{land}$ to avoid unnecessary landing and the following requirement: $(\square \neg \text{At.NoF1}) \land (\square \neg \text{At.NoF4}) \land (\bigcirc \text{At.A1}) \land (\bigcirc \text{At.B1})$.

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 6a. 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 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 update 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*.

**B. 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., [33], [35], [47]).

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 [34]. This follows the idea of sensor-based planning (e.g., [27]) in which a sensor (i.e., code) is provided to identify if a location correspond to the area to be covered.

Iterator-Based Planning works by providing a high-level iterator API (see Figure 7a) 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 if.next.inA?, and if it is (yes.next.inA) it should go to that location (see Figure 7b).

The system synthesised a controller for this mission in 0.5 s (using up to 14.0 MB RAM) and flew the UAV (we ran 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 6b. discrete cells were of $60 \times 60$ 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 6b 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 [48]. 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 6b and then specifying the new mission requirements, together with the standard transition requirement $\Theta_0$ and the rule (2) that only allows reconfig to happen when sensor.A is in its initial state. The environment state map $E'_\Theta$ 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.

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) that, when uploaded and hot-swapped, 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).

$$\Theta = \square (\text{reconfig} \Rightarrow \neg \text{SensingA})$$

$$\text{SensingA} = \{\text{is.next.inA}, \{\text{yes.next.inA}, \text{no.next.inA}\}, \bot\}$$

(2)

The UAV continues its mission (Figure 6b) covering B

(Plan_new).

C. Fire Monitoring

UAVs are used to aid firefighters by fire monitoring and tracking [49], [50]. A fire monitoring mission can be as simple as a fire lookout [51] 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 7b) 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. We synthesised (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 $30 \times 30$ m and a flight height of 70 m, totalling 1834 discrete locations. The resulting trajectory can be seen as Plan_old in Figure 6c.

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 V.B 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, $\Theta_1$ and $\Theta_2$ to restrict actions between specifications, and $\Theta_3$ to force a reset of the iterator before startNew (to guarantee full coverage of the region C).

$$\Theta_1 = \square ((\text{OldStopped} \land \neg \text{NewStarted}) \Rightarrow \neg \text{SenseOrMoveCmd})$$

$$\Theta_2 = \square (\text{reconfig} \Rightarrow \text{NewStarted})$$

$$\Theta_3 = \square (\text{startNew} \Rightarrow \text{Reset}), \text{Reset} = \langle \text{reset}, \text{has.next?}, \bot\rangle$$

(3)
where SenseOrMoveCmd is set to true with moving and sensing actions (e.g., takeOff, go, is.next.inAt?) and false with the rest.

The update controller was synthesised in 36.8 s (35.1 MB) and uploaded immediately. Figure 6 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 \( \Theta_3 \)). Further, to do reset it must not be moving (see Iterator requirements \( \text{[34]} \)), 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.

D. Unexpected Search & Rescue

Search & rescue scenarios are a common theme in robotics (e.g., \([39], \text{[52], [53]} \)) and the flying of the example in Section II shows 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 V-A synthesised in 0.4 s (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:

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

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:

\[
\square ((\text{OldStopped} \land \neg \text{NewStarted}) \implies \neg (\Gamma \setminus \{\text{low.height}, \text{high.height}\})) ,
\]

where the set \( \Gamma \) holds all the controllable actions. Update synthesis time totalled 15 s (54.6 MB).

VI. DISCUSSION AND FUTURE WORK

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 \([8] \) are not supported).

Many missions of interest in this domain are multi-vehicle \([47] \). One potential limitation of the approach is that discrete event controller synthesis may not scale to large missions. However, techniques such as Iterator-Based Planning \([44] \) have shown that missions for hundreds of thousands of discrete locations are tractable. Although we extended DCU for the purpose of typical recurring requirements in robotic missions, a comprehensive study of extensions to DCU to account for more complex liveness requirements is still needed and would be interesting to develop in future work.

VII. 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 hot-swapping 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] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litouin, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel et al., “Software engineering for self-adaptive systems: A second research roadmap,” in Software Engineering for Self-Adaptive Systems II. Springer, 2013, pp. 1–32.

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

[3] P. Hosek and C. Cadar, “Safe software updates via multi-version execution,” in Proceedings of the 2013 International Conference on Software Engineering, ser. ICSE ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 612–621.

[4] H. Chen, J. Yu, C. Hang, B. Zang, and P.-C. Yew, “Dynamic software updating using a relaxed consistency model,” Software Engineering, IEEE Transactions on, vol. 37, no. 5, pp. 679–694, Sept 2011.

[5] D. Gupta, P. Jalote, and G. Barua, “A formal framework for on-line software version change,” IEEE Trans. Software Eng., vol. 22, no. 2, pp. 120–131, 1996.

[6] J. Kramer and J. Magee, “The evolving philosophers problem: Dynamic change management,” IEEE Trans. Softw. Eng., vol. 16, no. 11, pp. 1293–1306, Nov. 1990.

[7] F. Banno, D. Marletta, G. Pappalardo, and E. Tramontana, “Handling consistent dynamic updates on distributed systems,” in Computers and Communications (ISCC), 2010 IEEE Symposium on, June 2010, pp. 471–476.

[8] H. Tajalli, J. García, G. Edwards, and N. Medvidovic, “Plasma: A plan-based layered architecture for software model-driven adaptation,” in Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ser. ASE ‘10. New York, NY, USA: ACM, 2010, pp. 467–476.

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

[10] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, “Specifying and verifying the correctness of dynamic software updates,” in Proceedings of the 4th International Conference onVerified Software: Theories, Tools, Experiments, ser. VSTTE’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 278–289.

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

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