

A Temporal Logic-Based Planning and Execution Monitoring System

Jonas Kvarnström and Fredrik Heintz and Patrick Doherty

Department of Computer and Information Science, Linköpings universitet
SE-581 83 Linköping, Sweden
{jonkv, frehe, patdo}@ida.liu.se

Abstract

As no plan can cover all possible contingencies, the ability to detect failures during plan execution is crucial to the robustness of any autonomous system operating in a dynamic and uncertain environment. In this paper we present a general planning and execution monitoring system where formulas in an expressive temporal logic specify the desired behavior of a system and its environment. A unified domain description for planning and monitoring provides a solid shared declarative semantics permitting the monitoring of both global and operator-specific conditions. During plan execution, an execution monitor subsystem detects violations of monitor formulas in a timely manner using a progression algorithm on incrementally generated partial logical models. The system has been integrated on a fully deployed autonomous unmanned aircraft system. Extensive empirical testing has been performed using a combination of actual flight tests and hardware-in-the-loop simulations in a number of different mission scenarios.

Introduction

Now and then, things will go wrong. This is both a fact of life and a fundamental problem in any robotic system operating autonomously or semi-autonomously in the real world. Like humans, robust systems must be able to take this into account and detect when events do not occur according to plan, regardless of whether this is the result of mechanical problems, incorrect assumptions about the world, or interference from other agents.

Some of these contingencies are best monitored in the low-level implementation of an action, especially when tight integration with control loops is necessary. However, much can be gained by complementing this with a higher level *execution monitor*. In addition to simplifying the specification of conditions that span multiple actions, this also improves the modularity of the execution system and avoids duplication of the necessary functionalities. Using an expressive declarative specification allows monitor conditions to be reasoned about at a higher level of abstraction.

In this paper, we present a general task planning and execution monitoring system based on the use of Temporal Action Logic (TAL, Doherty and Kvarnström 2008), a logic

for reasoning about action and change. TAL has already been used as the semantic basis for TALplanner (Doherty and Kvarnström 2001; Kvarnström 2005). We show how formulas in the same logic can be used to specify the desired behavior of an autonomous system and its environment, allowing easy specification of complex temporal conditions. An integrated domain description is used for both planning and monitoring, where conditions to be monitored may be global or specific to a particular plan operator. Some of these conditions can be automatically extracted from a plan. During execution, the DyKnow knowledge processing middleware (Heintz and Doherty 2004; 2006) incrementally builds a partial logical model representing the actual development of the system and its environment over time. The execution monitor uses this model together with a progression algorithm for prompt failure detection.

The planning and monitoring system is fully general and can be applied to a wide class of autonomous systems. It has currently been integrated and extensively tested on a fully deployed rotor-based unmanned aerial vehicle (UAV) system (Doherty 2004; 2005) using actual flight tests as well as hardware-in-the-loop simulations.

An Emergency Service Scenario

A number of very challenging scenarios have been used to drive both theoretical and applied research in our long-term UAV project. In this paper, we focus on a scenario where multiple UAVs are used to effectively aid the emergency services in a disaster situation such as the tsunami catastrophe in December 2004. The highest priorities in the initial stages of this disaster included searching for survivors in isolated areas where road systems had become inaccessible and providing relief in the form of food, water and medical supplies.

In the first phase of this mission, one or more UAVs cooperatively scan large regions generating a saliency map pinpointing potential victims. Our approach, which uses a combination of thermal and color cameras, has been successfully tested using two UAVs in an urban environment (Doherty and Rudol 2007; Rudol and Doherty 2008). In the second phase, used as a running example throughout the paper, the saliency map is used as a basis for generating a logistics plan for several UAVs to deliver food, water and medical supplies. In addition to directly delivering boxes, a UAV can also load several boxes onto a supply carrier, increasing its

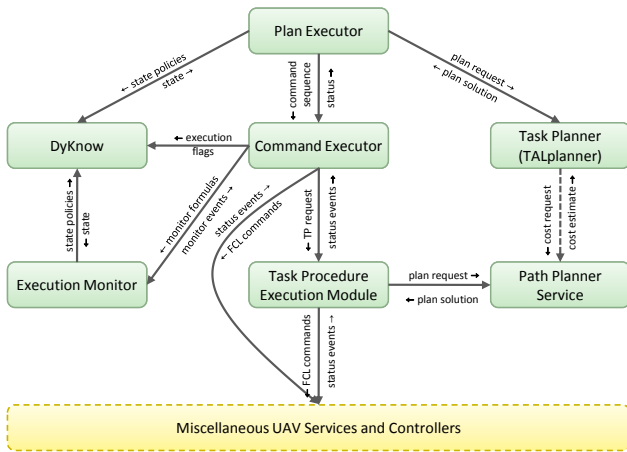


Figure 1: Task planning and execution monitoring overview

transportation capacity manyfold over longer distances.

A winch and electromagnet permitting a UAV to handle boxes without human assistance is under development. In the mean time, the second phase has been implemented and tested in simulation with hardware in the loop, using the same flight control software as the physical UAV. Faults can be intentionally injected or occur spontaneously due to the fact that our simulation environment incorporates rigid body dynamics, leading to effects such as boxes bouncing and rolling away should they be dropped.

A Planning and Monitoring System

Our approach to execution monitoring builds on the use of *monitor formulas* in an expressive temporal logic for declaratively specifying the desired behavior of an autonomous system and its environment over time. Monitor formulas are similar to temporal control formulas guiding forward-chaining search in planners such as TALplanner (Kvarnström 2005), but describe conditions on the actual state sequence resulting from plan execution rather than the state sequence predicted using operator definitions.

A unified domain specification for planning and monitoring supports the specification of global and operator-specific monitor formulas. Global formulas define domain-specific conditions that must hold over the entire execution of any plan. Operator-specific formulas must be satisfied starting when an instance of the associated operator is executed. This facilitates the use of context-dependent conditions, such as the fact that when a UAV collects a box, the action should succeed within a specific time interval, and the box should remain attached until explicitly detached. The fact that the box is explicitly detached can be referred to using a set of execution flags added to the logical model allowing a formula to determine what actions are currently executing.

System Overview. Figure 1 shows the relevant part of the UAV software architecture associated with task planning, plan execution and execution monitoring.

An autonomous system intended to perform complex missions requires timely data, information and knowledge pro-

cessing on many levels of abstraction. Low-level quantitative sensor data must be processed in multiple steps to eventually generate qualitative data structures which are grounded in the world and can be interpreted as knowledge by higher level deliberative services. *DyKnow*, a knowledge processing middleware framework, provides this service (Heintz and Doherty 2004; 2006).

Given a mission request, the *plan executor* calls *DyKnow* to acquire essential context-dependent information about the current state of the world and the UAV’s own internal state. Together with domain and goal specifications related to the current mission, this information is fed to *TALplanner*, which generates a high-level plan for the given goal. The domain specification also contains information about conditions to be monitored during plan execution, which is instantiated by *TALplanner* and included in the final plan.

The plan executor translates the high-level plan into lower level command sequences, leaving monitor conditions intact. The result is sent to the *command executor*, which is responsible for controlling the UAV, either by directly using its lowest level Flight Command Language (FCL) interface or by using *task procedures* (TPs). A TP is a high-level procedural execution component which provides a computational mechanism for achieving different robotic behaviors by using deliberative services and traditional control components in a highly distributed and concurrent manner. Flight-related TPs also use a *path planning service* (Wzorek et al. 2006; Wzorek and Doherty 2007) supporting several different path planning algorithms.

When execution starts, all global monitor formulas are immediately sent by the command executor to the *execution monitor*. Similarly, operator-specific monitor formulas are sent as execution of a particular operator instance begins.

The monitor subscribes to the necessary state information from *DyKnow*, providing a *state policy* describing desired sample rates, interpolation settings for use in case of missing values, and similar requirements on the state stream to be generated. As states are received, a progression algorithm is used to detect formula violations, which are immediately signaled to the command executor. The system can then perform a context-dependent recovery procedure taking information about the violated formula into account.

Domain Definition. Though this approach is not bound to a specific language, our implementation uses a domain description language based on the TAL (Temporal Action Logics) family of logics for reasoning about action and change (Doherty and Kvarnström 2008). Below, we describe the most pertinent aspects of the unified UAV domain definition.

TAL uses an order-sorted type system with a hierarchy of types. In the UAV domain, the type *LOCATABLE* represents objects having specific world coordinates. It has the subtypes *UAV* and *CARRYABLE*, the latter of which has the subtypes *BOX* and *CARRIER*. *CARRIER-POSITION* represents obstacle-free positions where a carrier may be placed.

Reasonable estimates of distances and timing are essential in order to predict mission completion times and fuel usage and to ensure no UAV has to wait unnecessarily for another. We therefore use a metric coordinate system rep-

represented by the $xpos(LOCATABLE)$ and $ypos(LOCATABLE)$ features (state variables), taking values from a finite domain FP of fixed-point numbers. Altitude is modeled using the $altitude(UAV)$ feature, also taking values from FP. The boolean feature $attached(UAV, CARRYABLE)$ holds if a certain UAV has attached its electromagnet to a certain carryable. Goals are specified in terms of boxes being sufficiently close to given coordinates.

The need to support monitoring and replanning after a failure serves to move the domain definition closer to the real world. For example, since boxes may be dropped, the common assumption that all objects are neatly placed must be withdrawn. Therefore $on-carrier(BOX, CARRIER)$ has a three-valued domain where a box may be definitely correctly placed on the carrier, definitely not on the carrier, or possibly misplaced (where it could block the electromagnet from attaching to the carrier, or could fall off when the carrier is lifted). Lifting a carrier with potentially misplaced boxes is not permitted. These cases are identified using object coordinates together with the approximate size of each carryable and the minimum safety distances between them.

Operators. We choose a comparatively fine-grained action model, where a UAV can align itself for attachment using $adjust-for-attach$, and then either $attach-box$ or $attach-carrier$. After a subsequent $climb-for-flying-with$, which reels in the winch and climbs to the standard flight altitude, it can fly the carryable to another location, $adjust-for-detach$, and either $detach-box$ or $detach-carrier$. After finishing with a $climb-for-flying-empty$ action, the UAV is free to pursue other goals. Though a more coarse-grained model is also possible, the fine-grained model facilitates replanning and simplifies the specification of monitor formulas applicable during a certain phase such as attachment or flight.

Five different *fly* operators are used: $fly-empty-to-box$, $fly-empty-to-carrier$, $fly-box-to-carrier$, $fly-box-to-goal$, and $fly-carrier$. This also simplifies the specification of control and monitor formulas, as the action of flying without cargo to pick up a box is appropriate in different circumstances than flying with a box to place it on a carrier and requires different conditions to be monitored.

Though operator parameters are generally not relevant for the purpose of this paper, an example is in order. The operator $fly-empty-to-carrier(uav, fromx, fromy, carrier, tox, toy)$ requires a UAV and its coordinates, plus a destination carrier and its coordinates, as parameters. Note that if one models an area of 10000 meters square at a resolution of 1 meter, the operator has 10^{16} ground instances, even with only a single UAV and a single carrier. Obviously, TALplanner does not generate all ground instances.

Control Formulas in TALplanner

TALplanner uses forward-chaining search guided by temporal control formulas that must be satisfied by the final plan.

In the sequential case, each search node corresponds to an executable action sequence that can be predicted to generate a finite state sequence $[s_0, s_1, \dots, s_m]$ if executed in the initial state s_0 . The corresponding (infinite) TAL interpretation cannot be constructed by repeating s_m indefinitely

($[s_0, s_1, \dots, s_m, s_m, \dots]$), as this would correspond to the assumption that no further actions can be performed. The possibility of future changes is properly taken into account by constructing a *partial* interpretation \mathcal{I} where feature values up to the time of state s_m are completely determined, after which no information is available. If ϕ is a control formula and $\mathcal{I} \models \neg\phi$, then any descendant must also violate the control formula: Adding a new action leaves the “past” intact while adding information about one or more new states, which cannot cause previous conclusions to be retracted. The planner can then reject the node and backtrack. How to test this efficiently, and how to adapt this process for concurrency, is discussed in Kvarnström (2005).

The following two examples use the TAL macro language $\mathcal{L}(ND)$, which can be translated into a first-order base language $\mathcal{L}(FL)$. In $\mathcal{L}(ND)$, $[\tau] \phi$ means that ϕ holds at time τ , and $f \hat{=} \omega$ indicates that the feature f has the value ω .

Example 1 (Global Control) *In the UAV domain, there are two acceptable reasons to move a box: It is not sufficiently close to its goal coordinates, or its location must temporarily be freed for other purposes, possibly because it is too close to a carrier position. This condition is simplified and modularized using several feature macros defined in terms of basic features: $close-enough-to-goal(box)$, $need-to-move-temporarily(box)$, and $is-at(locatable, x, y)$.*

$$\forall t, box, x, y. [t] is-at(box, x, y) \rightarrow [t+1] is-at(box, x, y) \vee$$

$$[t] \neg close-enough-to-goal(box) \vee$$

$$[t] need-to-move-temporarily(box) \quad \blacksquare$$

Example 2 (Operator-Specific Control) *A UAV should not prepare to attach to a carrier if there are potentially misplaced boxes closer than the designated safety distance. This can be specified as operator-specific control, which is similar to a precondition and has access to all operator parameters. For $adjust-for-attach$, start is the invocation time-point and carryable the box or carrier to be attached.*

$$[start] \forall carrier, box.$$

$$carrier = carryable \wedge too-close(box, carrier) \rightarrow$$

$$on-carrier(box, carrier) \hat{=} \textit{correctly placed} \quad \blacksquare$$

Execution Monitor Formulas

Monitor formulas and control formulas are quite similar in intent, expressing constraints on permitted state sequences. However, since they are applied in quite different circumstances, different evaluation mechanisms are appropriate.

Control formulas constrain the predicted development of the world during the planning phase. This development is already limited by the possible ways of composing actions in a plan, which enables the use of an operator-specific state transition analysis improving efficiency (Kvarnström 2005).

Monitor formulas describe conditions on the *actual* development of the world during execution. A progression algorithm (Bacchus and Kabanza 1998) applied to a state sequence incrementally generated by DyKnow can provide better performance in this situation, where unforeseen and unpredictable events and failures must be taken into account. Progression is expressed more naturally for formulas using relative time, where each formula is evaluated relative to a “current” timepoint. Thus we follow the TAL strategy of

adapting the $\mathcal{L}(\text{ND})$ macro language for the application at hand and introduce the macros U (until), \diamond (eventually) and \square (always), inspired by temporal modal logics such as MTL (Alur and Henzinger 1990). By convention, time is measured in milliseconds.

Definition 1 (Monitor Formula) A monitor formula is one of the following:

- $f \hat{=} \omega$, $\omega = \omega'$,
- $\phi \text{U}_{[\tau, \tau']} \psi$, $\diamond_{[\tau, \tau']} \phi$, $\square_{[\tau, \tau']} \phi$, or
- a combination of monitor formulas using the standard logical connectives and quantification over values,

where f is a feature term, ω and ω' are value terms, τ and τ' temporal terms, and ϕ and ψ monitor formulas. We allow the shorthand notation f for boolean features, meaning $f \hat{=} \text{true}$, and define $\phi \text{U} \psi \equiv \phi \text{U}_{[0, \infty)} \psi$, $\diamond \phi \equiv \diamond_{[0, \infty)} \phi$, and $\square \phi \equiv \square_{[0, \infty)} \phi$. ■

The semantics of monitor formulas is defined in terms of a translation into the TAL base language $\mathcal{L}(\text{FL})$, preserving the common semantic ground for planning and monitoring. The following conditions are satisfied by the translation:

- The formula $\phi \text{U}_{[\tau, \tau']} \psi$ (“until”) holds at time t iff ψ holds at some time $t' \in [t + \tau, t + \tau']$ and ϕ holds until then (at all timepoints in $[t, t')$, which may be an empty interval).
- The formula $\diamond_{[\tau, \tau']} \phi$ (“eventually”) is equivalent to $\top \text{U}_{[\tau, \tau']} \phi$ and holds at t iff ϕ holds at some timepoint $t' \in [t + \tau, t + \tau']$.
- The formula $\square_{[\tau, \tau']} \phi$ is equivalent to $\neg \diamond_{[\tau, \tau']} \neg \phi$ and holds at t iff ϕ holds at all timepoints $t' \in [t + \tau, t + \tau']$.

Definition 2 (Translation of Monitor Formulas) Let $\bar{\tau}$ be a temporal term and γ be a monitor formula intended to be evaluated at $\bar{\tau}$. The following translation yields an equivalent formula in $\mathcal{L}(\text{ND})$ without tense operators.

$$\begin{aligned} TM(\bar{\tau}, \mathcal{Q}x.\phi) &\stackrel{\text{def}}{=} \mathcal{Q}x.TM(\bar{\tau}, \phi), \text{ where } \mathcal{Q} \text{ is a quantifier} \\ TM(\bar{\tau}, \phi \otimes \psi) &\stackrel{\text{def}}{=} TM(\bar{\tau}, \phi) \otimes TM(\bar{\tau}, \psi), \text{ where } \otimes \text{ is} \\ &\quad \text{a binary connective} \\ TM(\bar{\tau}, \neg \phi) &\stackrel{\text{def}}{=} \neg TM(\bar{\tau}, \phi) \\ TM(\bar{\tau}, f \hat{=} v) &\stackrel{\text{def}}{=} [\bar{\tau}] f \hat{=} v \\ TM(\bar{\tau}, \gamma) &\stackrel{\text{def}}{=} \gamma, \text{ where } \gamma \text{ has no tense operators} \\ TM(\bar{\tau}, \phi \text{U}_{[\tau, \tau']} \psi) &\stackrel{\text{def}}{=} \exists t[\bar{\tau} + \tau \leq t \leq \bar{\tau} + \tau' \wedge \\ &\quad TM(t, \psi) \wedge \forall t'[\bar{\tau} \leq t' < t \rightarrow TM(t', \phi)]] \\ TM(\bar{\tau}, \square_{[\tau, \tau']} \phi) &\stackrel{\text{def}}{=} \forall t[\bar{\tau} + \tau \leq t \leq \bar{\tau} + \tau' \rightarrow TM(t, \phi)] \\ TM(\bar{\tau}, \diamond_{[\tau, \tau']} \phi) &\stackrel{\text{def}}{=} \exists t[\bar{\tau} + \tau \leq t \leq \bar{\tau} + \tau' \wedge TM(t, \phi)] \end{aligned}$$

The TAL translation function Trans from the $\mathcal{L}(\text{ND})$ to $\mathcal{L}(\text{FL})$ is extended for tense monitor formulas γ by defining $\text{Trans}(\gamma) = \text{Trans}(TM(0, \gamma))$. ■

Global monitor formulas are similar to global control formulas in TAL, expressing conditions that should be monitored throughout the execution of a plan.

Example 3 Suppose that a UAV supports a maximum continuous power usage of M , but can exceed this by a factor of f for up to τ units of time, if this is followed by normal power usage for a period of length at least τ' . The following

formula can be used to detect violations of this specification:

$$\square \forall uav. (\text{power}(uav) > M \rightarrow \text{power} < f \cdot M \text{U}_{[0, \tau]} \square_{[0, \tau']} \text{power}(uav) \leq M) \quad \blacksquare$$

Operator-specific formulas are not activated before plan execution but before the execution of a particular *step* in a plan, providing the ability to contextualize a monitor condition relative to a particular action. An operator-specific formula can also directly refer to the arguments of the associated operator. Note that while these conditions are triggered by the invocation of an operator, they can express conditions on the development of the world beyond the end of that operator.

Example 4 As attaching to a box may fail, the success of $\text{attach-box}(uav, \text{box}, x, y)$ should be monitored. The following operator-specific formula uses the first two arguments:

$$\diamond_{[0, 5000]} \square_{[0, 1000]} \text{attached}(uav, \text{box})$$

Within 5000 ms, the box should be attached. It should remain attached for at least 1000 ms, to detect problems during the attachment phase, where the electromagnet might briefly attach to the box but ultimately fail. ■

Operator-specific monitor formulas are instantiated with the parameters of the associated action. For example, $\text{attach-box}(\text{heli1}, \text{bx7}, 127.52, 5821.23)$ is annotated with the formula $\diamond_{[0, 5000]} \square_{[0, 1000]} \text{attached}(\text{heli1}, \text{bx7})$.

Execution Flag Features

The power of monitor formulas can be extended further by also giving introspective access to certain information about the plan execution state, in addition to the world state. For example, one might like to state that once a carrier has been attached to the UAV, it should remain attached until the UAV intentionally detaches it, that is, *until the corresponding detach action is executed*. Similarly, perhaps a certain fact should hold during the execution of an action, or an effect should be *achieved during* the execution of an action.

We therefore introduce *execution flags*, standard parameterized boolean features that hold exactly when the corresponding operator is being executed with a specific set of arguments. By convention, an execution flag is named by prepending “executing-” to the name of the corresponding operator. For example, attach-box is associated with the executing- attach-box flag, which takes a subset of the operator’s parameters as defined in the domain description. This flag is part of the state information extracted by DyKnow.

Example 5 The fact that $\text{climb-for-flying-empty}(uav)$ succeeds in ascending to a sufficient flight altitude A can be monitored using the operator-specific formula $\text{executing-climb-for-flying-empty}(uav) \text{U altitude}(uav) \geq A$. ■

When the operator is obvious from context, the shorthand notation EXEC is used to refer to its associated execution flag feature with the default parameters. Using this notation, Example 5 is written as $\text{EXEC U altitude}(uav) \geq A$.

Example 6 The more specific formula $(\text{rpm}(uav) \geq T \text{U altitude}(uav) \geq A) \vee (\text{EXEC U altitude}(uav) \geq A)$ is only violated if the engine RPM drops below a threshold T before success, indicating engine trouble. Note that low RPM

is in itself not sufficient to trigger the formula, if the operator eventually succeeds. Additional monitors can be used to model other causes of failure. ■

Example 7 The *attach-box*(uav, box, x, y) operator should succeed within 5000 ms, after which the box should remain attached until intentionally detached. This can be expressed using the following operator-specific formula:

executing-attach-box(uav, box) $U_{[0,5000]}$
 (attached(uav, box) U executing-detach-box(uav, box)) ■

Checking Monitor Conditions using Progression

Inspired by Ben Lamine and Kabanza (2002), we use a formula progression algorithm Pr to incrementally detect monitor formula violations during plan execution. By definition, ϕ holds in $[s_0, s_1, \dots, s_n]$ iff $Pr(\phi, s_0)$ holds in $[s_1, \dots, s_n]$. Thus, as soon as a single state s_0 has been created from sensory inputs, the progression algorithm can be applied to evaluate those parts of ϕ that depend on this state.

If the information provided by s_0 is sufficient to determine that the formula must be violated regardless of the future development of the world, $Pr(\phi, s_0)$ will return \perp (false), ensuring prompt failure detection. For example, this will happen as soon as the formula $\Box \text{speed} < 50$ is progressed through a state where $\text{speed} \geq 50$. Otherwise, progression returns a new and potentially different formula to be progressed again when the next state is available.

As states are not first class entities in TAL, the progression algorithm below is given a TAL interpretation and a timepoint corresponding to a state. The algorithm also takes advantage of regularities in state sampling periods: If samples arrive every m timepoints and all lower temporal bounds of tense operators in a formula ϕ are multiples of m' , we progress ϕ through $n = \text{gcd}(m, m')$ timepoints in a single step. The value of n is provided to the progression procedure defined below. This permits the temporal unit to be decoupled from the sample rate while at the same time retaining the standard TAL-based semantics, where states exist at every discrete timepoint. For example, suppose a timepoint corresponds to 1 ms. If samples arrive every 100 ms, the formula $\Diamond_{[0,3037]} \phi$ can be progressed through $\text{gcd}(100, 0) = 100$ timepoints in one step, while $\Diamond_{[40,3037]} \phi$ can be progressed through $\text{gcd}(100, 40) = 20$ timepoints.

Let n be a progression step in timepoints, ϕ a monitor formula where lower bounds are multiples of n , τ a numeric timepoint, and \mathcal{I} a TAL interpretation. Then, $Pr(\phi, \tau, n, \mathcal{I})$ holds at $\tau + n$ in \mathcal{I} iff ϕ holds at τ in \mathcal{I} . More formally, $\mathcal{I} \models \text{Trans}(\text{TM}(\tau, \phi))$ iff $\mathcal{I} \models \text{Trans}(\text{TM}(\tau + n, Pr(\phi, \tau, n, \mathcal{I})))$.

Definition 3 (Progression of Monitor Formulas) The following algorithm is used for progression. Special cases for \Box and \Diamond can also be introduced for performance.

```

1 procedure Pr( $\phi, \tau, n, \mathcal{I}$ )
2 if  $\phi = f(\bar{x}) \doteq v$ 
3   if  $\mathcal{I} \models \text{Trans}([\tau] \phi)$  return  $\top$  else return  $\perp$ 
4 if  $\phi = \neg\phi_1$  return  $\neg Pr(\phi_1, \tau, n, \mathcal{I})$ 
5 if  $\phi = \phi_1 \otimes \phi_2$  return  $Pr(\phi_1, \tau, n, \mathcal{I}) \otimes Pr(\phi_2, \tau, n, \mathcal{I})$ 
6 if  $\phi = \forall x. \phi //$  where  $x$  belongs to the finite domain  $X$ 
7   return  $\bigwedge_{c \in X} Pr(\phi[x \mapsto c], \tau, n, \mathcal{I})$ 
8 if  $\phi = \exists x. \phi //$  where  $x$  belongs to the finite domain  $X$ 

```

```

9   return  $\bigvee_{c \in X} Pr(\phi[x \mapsto c], \tau, n, \mathcal{I})$ 
10 if  $\phi$  contains no tense operator
11   if  $\mathcal{I} \models \text{Trans}(\phi)$  return  $\top$  else return  $\perp$ 
12 if  $\phi = \phi_1 U_{[\tau_1, \tau_2]} \phi_2$ 
13   if  $\tau_2 < 0$  return  $\perp$ 
14   elsif  $0 \in [\tau_1, \tau_2]$  return  $Pr(\phi_2, \tau, n, \mathcal{I}) \vee$ 
15     ( $Pr(\phi_1, \tau, n, \mathcal{I}) \wedge (\phi_1 U_{[\tau_1 - n, \tau_2 - n]} \phi_2)$ )
16   else return  $Pr(\phi_1, \tau, n, \mathcal{I}) \wedge (\phi_1 U_{[\tau_1 - n, \tau_2 - n]} \phi_2)$ 

```

The result of Pr is simplified using the rules $\neg\perp = \top$, $(\perp \wedge \alpha) = (\alpha \wedge \perp) = \perp$, $(\perp \vee \alpha) = (\alpha \vee \perp) = \alpha$, $\neg\top = \perp$, $(\top \wedge \alpha) = (\alpha \wedge \top) = \alpha$, and $(\top \vee \alpha) = (\alpha \vee \top) = \top$. Further simplification is possible using identities such as $\Diamond_{[0, \tau]} \phi \wedge \Diamond_{[0, \tau']} \phi \equiv \Diamond_{[0, \min(\tau, \tau'')]} \phi$. ■

Automatic Generation of Monitor Formulas

Using a single logical formalism for both planning and monitoring provides a suitable basis for automatic generation of monitor formulas. Automation can help eliminate many potential sources of inconsistencies and inaccuracies, but at the same time, a certain degree of selectivity is required. Some violations are not fatal, some information about the environment may be expensive or difficult to sense, and sensing may require actions that interfere with normal mission operations. Also, the introduction of a richer and more detailed domain model should not automatically lead to heavier use of sensors. For these reasons, each declaration that can be used to extract monitor conditions can be annotated with a flag stating whether monitoring is required. This increases the benefits of automatic formula generation by keeping the control in the hands of the domain designer.

Preconditions. An operator precondition ϕ can be directly used as a monitor formula, testing whether ϕ holds when the operator is invoked.

Preval conditions. A prevail condition ϕ is similar to a precondition, but must hold throughout the execution of the action. The formula $(\text{EXEC} \wedge \phi) U \neg\text{EXEC}$ can be used.

Effects. The formula $\text{EXEC} U \phi$ expresses the condition that the effect ϕ is achieved at some time during the execution of an action while $\text{EXEC} U (\neg\text{EXEC} \wedge \phi)$ ensures that ϕ still holds at the *end* of the action.

Temporal Constraints. An operator can be associated with constraints on the duration of its execution. Such constraints can be used to generate monitor formulas constraining the amount of time that can pass before $\neg\text{EXEC}$ holds.

Causal Links. A precondition might depend on earlier effects. For example, detaching a box at its destination requires that the box is still attached, which was an effect of an earlier attach-box action. Effect and precondition monitors provide a partial solution, but do not monitor attachment during intervening flight actions. Causal link analysis in the complete plan detects the fact that attached(uav, box) is made true by attach-box, is required by detach-box, and is not altered by any action in between. The initial attach-box action can then automatically be annotated with the formula executing-attach-box(uav, box) U (attached(uav, box) U executing-detach-box(uav, box)), instantiated with the appropriate UAV and box.

Execution Monitoring with Imperfect Sensors

To be useful in an autonomous system, an execution monitor must be robust against a certain amount of sensor noise. Similarly, it should be possible to handle occasional dropouts, communication delays and values arriving out of order. This is especially important in distributed architectures, perhaps consisting of multiple UAV platforms together with a set of ground stations.

Careful state generation provides part of the solution. Temporary dropouts can sometimes be handled through extrapolation. Communication delays can be handled by delaying progression for a certain amount of time, at the cost of also delaying failure detection. Noise could be minimized through sensor value smoothing techniques and fusion of values from multiple sensors. However, the possibility of inaccuracies in the generated state sequence can in the general case never be completely eliminated.

Monitor formulas provide the required expressivity to take temporary inaccuracies into account, while making the necessary domain-dependent tradeoffs between false positives and delayed failure detection. For example, $\square \forall uav.speed(uav) \leq T$ means that the *sensed approximate* speed of a UAV should never exceed T . As a single observation above T might be a sensor error, one could instead require that the sensed speed never exceed the threshold in an interval of length τ , expressed as $\square \diamond_{[0,\tau]} speed(uav) \leq T$.

Since this formula would be satisfied by a single sample below the threshold every τ milliseconds, it might be considered too weak. An alternative would be to require an *interval* of length τ' where the UAV stays within the limits: $\square (speed(uav) > T \rightarrow \diamond_{[0,\tau]} \square_{[0,\tau']} speed(uav) \leq T)$.

Empirical Evaluation of the Progressor

Compared to a desktop computer, an autonomous system generally has considerably less processing power. As much of this power tends to be required for execution, a monitoring system must be able to run using quite limited resources.

Testing in the emergency services domain, both in flight tests and in hardware-in-the-loop simulation, indicates that the formulas we use can easily be handled using only a fraction of the 1.4 GHz Pentium M CPU onboard one of our UAVs. If there is a bottleneck, then it lies not in the use of formula progression but in retrieving and processing sensory data, which is necessary regardless of the specific approach being used for monitoring execution. Nevertheless, processing requirements for typical formulas should be quantified.

Two different formulas were used in the experiments. The first formula is $\square \diamond_{[0,1000]} p$, where p is a single feature, corresponding to the fact that p must never remain false for more than one second. The second formula is $\square (\neg p \rightarrow \diamond_{[0,1000]} \square_{[0,999]} p)$, corresponding to the fact that if p becomes false, then within 1000 ms, there must begin a period lasting at least 1000 ms where p is true.

For each formula we constructed two different state sequences corresponding to the best and worst case with regard to expansion of the formulas during progression. The sample period was 100 ms.

- **(true)** – p is always true. This is the best case in terms

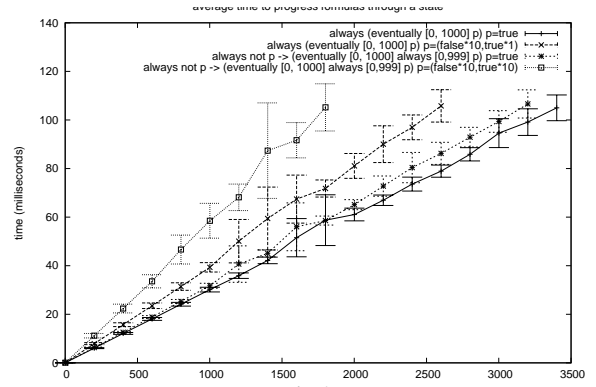


Figure 2: Average Progression Time

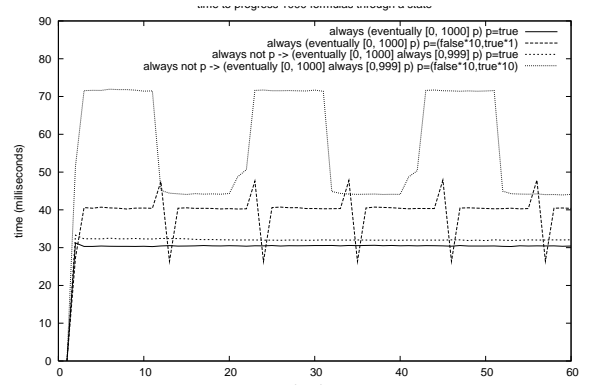


Figure 3: Progression Time per Iteration

of performance for both formulas. For the first formula, each time $\diamond_{[0,1000]} p$ is progressed through a state, it immediately “collapses” into \top . What remains to evaluate in the next state is the original formula, $\square \diamond_{[0,1000]} p$. For the second formula it is enough to check that p is not false and then return the original formula.

- **(false*10,true*1)** – the worst case for the first formula, where p remains false for 10 consecutive sampling periods (1000 ms), after which it is true in a single sample. The sequence then repeats. Had p remained false for 11 samples, both formulas would have been violated.
- **(false*10,true*10)** – the worst case for the second formula, where p is false during 10 samples and true during 10 samples, after which the sequence repeats.

The results from progressing multiple instances of each formula are shown in Figure 2. Given a sample period of 100 ms and full use of the CPU, approximately 2500 instances of the first formula or 1500 instances of the second formula can be used even with the worst case state sequence. Equivalently, 5% of the CPU is sufficient for 125 instances of the first formula or 75 instances of the second formula.

As progression returns a new and not necessarily identical formula, time requirements may vary across iterations. The average progression time must be sufficiently low, or the progressor will permanently fall behind. The *maximum*

progression time should also be sufficiently low, or the progressor will *temporarily* fall behind.

Figure 3 shows the precise time used for each state (iteration) when progressing 1000 formula instances. As can be seen, the time to progress a formula through its worst case state sequence essentially alternates between two plateaus. The lower plateau corresponds to p being true, while the higher plateau corresponds to p being false. Varying I and J in the generalized formulas $\Box \Diamond_{[0,I]} p$ and $\Box (\neg p \rightarrow \Diamond_{[0,I]} \Box_{[0,J-1]})$ will only affect the permissible lengths of these plateaus without affecting their levels. Average time requirements must be somewhere between these fixed plateaus, which is confirmed by experimental results.

Recovery from Failures

Any monitor formula violation signals a potential or actual failure from which the system must attempt to recover in order to achieve its designated goals. This is a complex topic, especially when combined with the stringent safety regulations associated with flying a UAV and the possibility of having time-dependent goals and constraints. For example, if **heli1** fails to deliver a box of medicine on time, **heli2** might have to be rerouted in order to meet a deadline. Therefore, our first iteration of the recovery system has not tackled incremental plan repair, even though this may be desirable in the long run. Instead, each monitor formula is associated with a *recovery operator* that can perform emergency recovery procedures if necessary, and possibly also adjust the UAV's assumptions about the world. For example, if the UAV fails to take off with a carrier, this information is used to contextually modify assumptions about how many boxes can be lifted at once. This is followed by gathering information about the current state from DyKnow and replanning. Since TALplanner is sufficiently fast, this does not adversely affect the execution of a fully autonomous mission.

Alternative Approaches

Many architectures that deal with both planning and execution focus entirely on recovery from detected problems by plan repair or replanning (Lemai and Ingrand 2004; Ambros-Ingerson and Steel 1988). These architectures usually assume that state variables are correctly updated and that plan operator implementations detect any possible failure in some unspecified manner, and thereby do not consider the full execution monitoring problem. A more elaborate and general approach is taken by Wilkins, Lee and Berry (2003) where a large set of different types of monitors are used in two different applications. However, monitors are procedurally rather than declaratively encoded.

In those architectures that do consider execution monitoring it is often an intrinsic part of an execution component rather than integrated with the planner (Fernández and Simmons 1998; Simmons and Apfelbaum 1998). The most common approach uses a predictive model to determine what state a robot should be in, continuously comparing this to the current state as detected by sensors (Chien et al. 2000; Washington, Golden, and Bresina 2000). However, the fact that a discrepancy between the current state and the pre-

dicted state has been detected does not necessarily mean that this discrepancy has a detrimental effect on the plan. Thus, one must take great care to distinguish essential deviations from unimportant ones, reducing the benefits of this approach. One should also take into account that not all predictable facts can be monitored, as a system may lack the necessary sensors and resources to simultaneously monitor all possible conditions. Even for those conditions that a system can detect, excessive monitoring may cause problems when the cost of information gathering is not negligible.

Another major weakness in these approaches is that only the current state is considered. Adapting ideas from model checking to be able to express constraints on sequences of states, Ben Lamine and Kabanza (2002) expressed the desired properties of a system in a temporal logical formalism. Whereas standard model checking tests such properties against a system model, their monitoring system tests them against an actual execution trace. Similar ideas have also been independently developed in the model checking community, where the problem of testing against a single execution trace is called path model checking (Markey and Raskin 2006), or runtime verification if the evaluation is done incrementally as the trace develops (Thati and Rosu 2005).

Though this provided part of the inspiration for this article, the work by Ben Lamine and Kabanza focuses on a reactive behavior-based architecture where the combined effects of a set of interactive behaviors is difficult to predict in advance. There, monitor formulas generally state global properties that cannot be monitored by internal behaviors, such as the fact that after three consecutive losses of communication, a given behavior must be active. A violation triggers an ad-hoc behavior that attempts to correct the problem. In comparison, our system is based on the use of planning, with an integrated domain description language. Formulas are not necessarily global, but can be operator-specific. We provide a selective mechanism for extracting monitor formulas through automated plan analysis and support recovery through replanning. DyKnow also gives us the possibility to provide attention focused state descriptions, where the execution monitor contextually subscribes only to those state variables that are required for progressing the currently active monitor formulas. Additionally, state sampling rates can be decided individually for each monitor formula. Combining this with the use of operator-specific monitor formulas that are only active when specific tasks are being performed ensures that state variables are only computed when strictly required, ensuring minimal usage of resources including sensors and computational power. This can be particularly beneficial in the case of state variables requiring image processing or other complex operations.

See Pettersson (2005) for an overview of systems related to execution monitoring.

Conclusions

We have presented a system where conditions to be monitored during plan execution are integrated into a declarative logic-based domain specification that also provides the necessary information for plan generation. Using the same expressive logic formalism and the same underlying semantics

for both planning and monitoring has several important advantages. For example, monitor conditions can be attached to specific plan operators, and plans can be analyzed to automatically extract conditions to be monitored. There are also significant benefits to lifting many aspects of monitoring and failure detection from the execution subsystem into a high-level declarative specification used by a distinct execution monitor subsystem. Perhaps most importantly, this serves to decouple action implementations, which by their very nature must be domain-specific, from the domain-independent functionality required for execution monitoring in any application. Thus, the planning and execution monitor system presented here is fully generic and can be applied to a large class of autonomous systems.

Currently, the system has been implemented and integrated into an unmanned aircraft system. Quantitative empirical testing shows that a large number of conditions can be monitored concurrently using a fraction of the computational resources available onboard a UAV. Extensive empirical testing has also been performed using a combination of actual flight tests and hardware-in-the-loop simulations in a number of challenging mission scenarios. For example, a variety of failure types have successfully been detected in an emergency services mission.

Based on a great deal of experience with our UAV systems, it is our strong belief that using logics as the basis for deliberative functionalities such as planning and monitoring simplifies the development of complex intelligent autonomous systems. Temporal Action Logic and its tense formula subset are highly expressive languages which are well suited for describing planning domains and for expressing the monitoring conditions we are interested in. Therefore we believe this approach provides a viable path towards even more sophisticated, capable, and robust autonomous systems.

Acknowledgements

This work is partially supported by grants from the Swedish Research Council (50405001, 50405002), the Swedish Aeronautics Research Council (NFFP4-S4203), the Swedish Foundation for Strategic Research (SSF) Strategic Research Center MOVIII, and CENIIT, the Center for Industrial Information Technology.

References

- Alur, R., and Henzinger, T. 1990. Real time logics: complexity and expressiveness. In *Fifth annual symposium on logic in computer science*.
- Ambros-Ingerson, J., and Steel, S. 1988. Integrating planning, execution and monitoring. In *Proc. AAAI*.
- Bacchus, F., and Kabanza, F. 1998. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence* 22.
- Ben Lamine, K., and Kabanza, F. 2002. Reasoning about robot actions: A model checking approach. In *Advances in Plan-Based Control of Robotic Agents*.
- Chien, S.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve the responsiveness of planning and scheduling. In *Proc. AAAI*.
- Doherty, P., and Kvarnström, J. 2001. TALplanner: A temporal logic-based planner. *AI Magazine* 22(3).
- Doherty, P., and Kvarnström, J. 2008. Temporal action logics. In Lifschitz, V.; van Harmelen, F.; and Porter, F., eds., *Handbook of Knowledge Representation*. Elsevier.
- Doherty, P., and Rudol, P. 2007. A UAV search and rescue scenario with human body detection and geolocalization. In *Proc. Australian Joint Conference on AI*.
- Doherty, P. 2004. Advanced research with autonomous unmanned aerial vehicles. In *Proc. KR*.
- Doherty, P. 2005. Knowledge representation and unmanned aerial vehicles. In *Proc. IEEE Conference on Intelligent Agent Technology*.
- Fernández, J. L., and Simmons, R. G. 1998. Robust execution monitoring for navigation plans. In *Proc. IROS*.
- Heintz, F., and Doherty, P. 2004. DyKnow: An approach to middleware for knowledge processing. *J. Intelligent and Fuzzy Systems* 15(1).
- Heintz, F., and Doherty, P. 2006. DyKnow: A knowledge processing middleware framework and its relation to the JDL data fusion model. *J. Intell. and Fuzzy Systems* 17(4).
- Kvarnström, J. 2005. *TALplanner and Other Extensions to Temporal Action Logic*. Ph.D. Dissertation, Linköpings universitet.
- Lemai, S., and Ingrand, F. 2004. Interleaving temporal planning and execution in robotics domains. In *Proc. AAAI*.
- Markey, N., and Raskin, J.-F. 2006. Model checking restricted sets of timed paths. *Theoretical Computer Science* 358(2-3).
- Pettersson, O. 2005. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems* 53(2).
- Rudol, P., and Doherty, P. 2008. Human body detection and geolocalization for UAV search and rescue missions using color and thermal imagery. In *Proc. IEEE Aerospace Conference*.
- Simmons, R., and Apfelbaum, D. 1998. A task description language for robot control. In *Proc. IROS*.
- Thati, P., and Rosu, G. 2005. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science* 113.
- Washington, R.; Golden, K.; and Bresina, J. 2000. Plan execution, monitoring, and adaptation for planetary rovers. *Electronic Transactions on Artificial Intelligence* 5(17).
- Wilkins, D.; Lee, T.; and Berry, P. 2003. Interactive execution monitoring of agent teams. *J. Artificial Intelligence Research* 18.
- Wzorek, M., and Doherty, P. 2007. A framework for reconfigurable path planning for autonomous unmanned aerial vehicles. *J. Applied Artificial Intelligence*. Forthcoming.
- Wzorek, M.; Conte, G.; Rudol, P.; Merz, T.; Duranti, S.; and Doherty, P. 2006. From Motion Planning to Control – A Navigation Framework for an Autonomous Unmanned Aerial Vehicle. In *Proc. Bristol Int'l UAV Systems Conf.*