Robotics, Temporal Logic and Stream Reasoning

Patrick Doherty[1]*, Fredrik Heintz[1], and Jonas Kvarnström[1]

Linköping University, Department of Computer and Information Sciences, S-581 83 Linköping, Sweden
patrick.doherty@liu.se, fredrik.heintz@liu.se, jonas.kvarnstrom@liu.se

Abstract

The area of AI robotics offers a set of fundamentally challenging problems when attempting to integrate logical reasoning functionality in such systems. The problems arise in part from the high degree of complexity in such architectures which include realtime behaviour, distribution, concurrency, various data latencies in operation and several levels of abstraction. For logic to work practically in such systems, traditional theorem proving, although important, is often not feasible for many of the functions of reasoning in such systems. In this article, we present a number of novel approaches to such reasoning functionality based on the use of temporal logic. The functionalities covered include automated planning, stream-based reasoning and execution monitoring.

1 Introduction

Logic has always played a central role in artificial intelligence. Robotics has been a central topic in AI from its inception. AI robotics focuses specifically on developing robotic platforms that exhibit intelligent behaviours. The basis for such behaviours is goal-directed and includes various forms of reasoning, either explicit or implicit.

This paper focuses on a number of logic-based functionalities used in very complex autonomous unmanned aircraft systems developed and deployed in the past decade [4]. The focus here will be on a temporal-logic based automated planner, TALplanner; an execution monitoring system; and a generic middleware component, DyKnow, for stream-based reasoning. Each functionality is based on the explicit use of temporal logic either as a specification language or as a novel form of soft realtime reasoner, or both.

The realtime nature of unmanned aircraft systems functioning in complex operational environments brings with it a set of unique challenges in the use of logic in such systems. Unmanned aircraft systems generally operate in both realtime and soft-realtime modes. Reasoning about both time and space are essential for successful goal achievement in the majority of operational environments. These systems are required to reason about their embedding environments, but also about their internal processes. The software architectures required for such systems are highly complex. They are distributed systems in terms of both hardware and software, and processes are concurrent and can be both synchronous and asynchronous. Data flow in the system has both realtime and soft-realtime requirements in addition to various latencies in the system.

If one is serious about designing and constructing robotic platforms that exhibit intelligent behaviour, there are a number of fundamental, open challenges of both a pragmatic and theoretical nature that need to be solved. Architecturally, such systems must combine realtime control behaviours, soft-realtime reactive behaviours and deliberative behaviours in one integrated system where these different behaviours

[*] This work is partially supported by The Swedish Research Council (VR) Linnaeus Center for Control, Autonomy, Decision-making in Complex Systems (CADICS), the ELLIIT network organization for Information and Communication Technology, the Swedish National Aviation Engineering Research Program NFFP6, SSF – the Swedish Foundation for Strategic Research (CUAS Project) and the EU FP7 project SHERPA, grant agreement 600958.

operate concurrently. There is a continual trade-off between the use of reactive and deliberative behaviours and these operate in the context of a realtime control kernel which, in the case of a UAV (Unmanned Aerial Vehicle), keeps it navigationally robust and in the air. Descriptions of such architectures are covered elsewhere [7], but are the basis for supporting the functionalities described here.

A related issue of fundamental importance is the principled management of dataflow in such architectures. This issue is often overlooked, but is central to the ability of a system to reason at a high level of abstraction in a semantically grounded and qualitative manner. Control, reactive and deliberative processes each place requirements on the flow of data input into these processes and data output delivered by these processes. Conceptually, such dataflow can be characterized as streams containing sequences of time-stamped data objects delivered at various sampling rates and used by various processes. These streams need to be specified, generated, merged, filtered and managed in many ways. Lurking here is a generic middleware functionality which is a prerequisite to powerful uses of stream-based reasoning in robotic architectures. In section 2 such a middleware service, DyKnow, is presented.

Another fundamental challenge is to close the gap between the low-level numeric data streams generated by sensors in a robotic platform and the high-level qualitative data streams required for deliberative processes. The issue here is generating semantically grounded symbol structure streams from sensed data. This is often called closing the sense-reasoning gap. It turns out that various processes require data at various levels of abstractions and at various latencies. For example, suppose a UAV wants to reason about vehicle behaviour on the ground using chronicle recognition systems and qualitative spatial reasoning systems. How would movement in a video camera image be translated in real-time into qualitative descriptions that can be used by deliberative components? In section 2, one approach to doing this successfully is described.

One important area of knowledge representation is cognitive robotics. One of the principal tools used to specify such systems is logics of action and change [5]. One such logic, TAL, is briefly described in section 3. It is a nonmonotonic linear temporal logic based on the idea of features and fluents. A fluent is a time indexed function returning the value of a particular feature at a point in time. A fluent is a formal representation of a stream which is an approximation to that fluent. A model for a theory in TAL is an aggregate of many fluents. This aggregate is equivalent to a stream of states. This is the central idea which relates logics of action and change to streams generated using DyKnow. DyKnow provides the ability to generate aggregates of streams in realtime. Such an aggregate is viewed as a model representing aspects of the embedding environment of a robotic system, or the internal behaviour of the robotic system itself. One can then base a realtime stream-based reasoning system on this connection by querying such streams with temporal formulas using a progression algorithm.

Generating plans to achieve goals is a central aspect of intelligent behaviour. For a UAV system, both task planning and motion planning is required to achieve goals. In section 3.1, a temporal logic based planner, TALplanner, is discussed. It is formally specified using TAL. The input to the planner is a logical theory in the form of a narrative. The output of the planner is an extended narrative that achieves the goals in question. Although temporal logic is used as basis for the planner, it is one of the most efficient task planners in existence. Plans themselves, whether partial or complete, can be reasoned about logically since they are logical theories in TAL.

In dynamic environments, plans have a tendency to break down. Replanning has to be done continually due to changes in the predicted behaviour of the surrounding environment. In order to replan, the planner must be made aware of the fact that something is wrong. In section 3.2, an execution monitoring system based on the use of streams and temporal logic is described. Since a plan is a logical theory, the incremental success of the plan's execution can be checked by incrementally querying an aggregated set of streams generated by DyKnow. The aggregated stream is a model which should satisfy the logical theory representing the plan. If it does not, a problem is identified and can be dealt with. The central idea is that DyKnow can generate temporal models on the fly and these models can be queried by checking

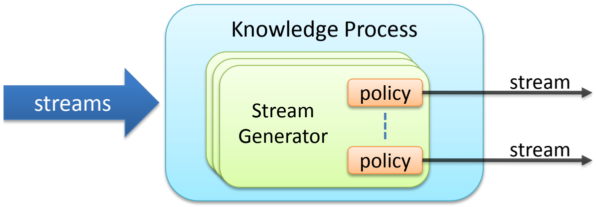

Figure 1: A prototypical knowledge process.

the satisfiability of a query specified as a temporal formula.

The functionality for querying temporal models is much more general. It can be viewed as a generic stream-based reasoning system integrated in a robotic system. Various queries can be made, not only for execution monitoring, but for querying in general. For example, safety and liveness conditions associated with the robust behavior of the robotic system can be continually checked in realtime. Queries to diagnose problems with the system can be represented as temporal formulas and checked against an aggregate of streams generated by DyKnow for just that purpose. Descriptions and details for each of these functionalities are presented in the remaining part of the paper.

2 DyKnow: Stream-Based Reasoning Middleware

The main purpose of DyKnow [10, 11] is to provide generic and well-structured middleware support for the generation of state, object, and event abstractions for the environments of complex systems. Such generation is done at many levels of abstraction beginning with low level quantitative sensor data and resulting in qualitative data structures which are grounded in the world and can be interpreted as knowledge.

DyKnow organizes the many levels of information and knowledge processing in a distributed robotic system as a network of *knowledge processes* connected by *streams* of time-stamped data or knowledge (Figure 1). Streams may for example contain periodic readings from sensors or sequences of query results from databases. Knowledge processes can use a *semantic integration* functionality to find streams based on their semantics relative to a common ontology. They can then process the streams by applying functions, synchronization, filtering, aggregation and approximation as they move to higher levels of abstraction. They often provide *stream generators* producing new streams satisfying *policies*, declarative specifications of desired stream properties. In this way, DyKnow supports conventional data fusion processes, but also less conventional qualitative processing techniques common in artificial intelligence.

For modelling purposes, the environment of a robotic system is viewed as consisting of physical and non-physical *objects* (such as *the UAV*, *car37* and *the entity observed by the camera*), *properties* associated with these objects, and *relations* between the objects. Properties and relations, such as the *velocity* of an object and the *distance between* two car objects, are called *features*. Since each feature can change values over time, it is associated a total function from time to value called a *fluent*.

Due to inherent limitations in sensing and processing, one cannot expect access to an actual fluent. Instead a *fluent stream* is an *approximation* of a fluent, containing a stream of *samples* of feature values at specific time-points. A collection of fluent streams corresponds directly to a temporal logical model.

A sample can either come from an observation of the feature or a computation which results in an estimation of the value at the particular time-point, called the *valid time*. The time-point when a sample is made available or added to a fluent stream is called the *available time*. A fluent stream has certain properties such as start and end time, sample period and maximum delay. These properties are specified

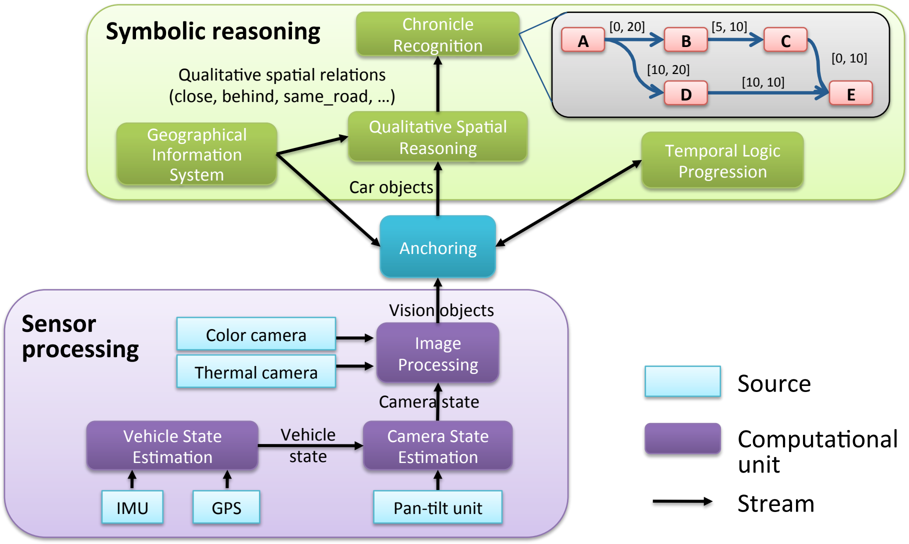

Figure 2: Potential organization of the incremental processing required for a traffic surveillance task.

by a declarative policy which describes constraints on the fluent stream.

For example, the position of a car can be modeled as a feature. The true position of the car at each time-point during its existence would be its fluent and a particular sequence of observations or estimations of its position would be a fluent stream. There can be many fluent streams all approximating the same fluent.

DyKnow also supports *state generation*, creating an approximation of the state at a given time using the information that has propagated through the distributed system so far. For example, two streams representing “speed of car$_1$” and “position of car$_1$” can be synchronized into a single fluent stream containing $\langle speed, position \rangle$ *tuples*, called *states*, whose values one cannot necessarily necessarily measure exactly simultaneously but must be estimated at a common timepoint.

A Traffic Monitoring Example. Figure 2 shows how part of the incremental processing required for a UAV traffic surveillance task can be organized as a set of knowledge processes.

At the lowest level, a *helicopter state estimator* uses data from an *inertial measurement unit* (IMU) and a *global positioning system* (GPS) to determine the position and attitude of the UAV. A *camera state estimator* uses this together with the state of the *pan-tilt unit* on which the cameras are mounted. The *image processing component* uses the camera state to determine where the camera is pointing. Video streams from *color* and *thermal cameras* can then be analyzed to generate *vision percepts* representing hypotheses about physical objects, including approximate positions and velocities.

Symbolic formalisms require a consistent assignment of symbols, or identities, to the physical objects being reasoned about and the sensor data received about the objects. Image analysis may provide a partial solution, but changing visual conditions or objects temporarily being out of view lead to problems that image analysis cannot necessarily handle. The *anchoring system* therefore uses *progression* of formulas in a metric temporal logic to incrementally evaluate potential hypotheses about the observed objects [12]. The anchoring system also assists in object classification and in the extraction of higher level attributes of an object. For example, a *geographic information system* can be used to determine whether an object is currently on a road or in a crossing. Such attributes can in turn be used to derive relations *between* objects, including *qualitative spatial relations* such as $\mathsf{beside}(car_1, car_2)$ and

$\mathsf{close}(car_1, car_2)$. Concrete events corresponding to changes in such attributes and relations allow the *chronicle recognition system* to determine when higher-level events such as reckless overtakes occur.

Related Work. To the best of our knowledge there does not really exist any other system which provides similar stream reasoning functionality. The KnowRob system [20] is probably the closest match with its sophisticated and powerful knowledge processing framework. However, it does not support reasoning over streaming information and the support for temporal reasoning is limited.

3 Temporal Action Logic

Temporal Action Logic, TAL, is a well-established non-monotonic logic for representing and reasoning about actions [5, 8]. Here a limited subset of TAL is described and we refer to [5] for further details.

TAL provides an extensible macro language, $\mathscr{L}$(ND), that allows reasoning problems to be specified at a higher abstraction level than plain logical formulas. The basic ontology includes parameterized features $f(\bar{x})$ that have values v at specific timepoints t, denoted by $[t] f(\bar{x}) \hat{=} v$, or over intervals, $[t,t'] f(\bar{x}) \hat{=} v$. Incomplete information can be specified using disjunctions of such facts. Parameterized actions can occur at specific intervals of time, denoted by $[t_1,t_2] A(\bar{x})$. To reassign a feature to a new value, an action uses the expression $R([t] f(\bar{x}) \hat{=} v)$. Again, disjunction can be used inside $R()$ to specify incomplete knowledge about the resulting value of a feature. The value of a feature at a timepoint is denoted by $value(t,f)$.

The logic is based on scenario specifications represented as *narratives* in $\mathscr{L}$(ND). Each narrative consists of a set of statements of specific types, including *action type specifications* defining named actions with preconditions and effects. The basic structure, which can be elaborated considerably [5], has the form $[t_1,t_2] A(\bar{v}) \rightsquigarrow (\Gamma_{pre}(t_1,\bar{v}) \to \Gamma_{post}(t_1,t_2,\bar{v})) \land \Gamma_{cons}(t_1,t_2,\bar{v})$ stating that if the action $A(\bar{v})$ is executed during the interval $[t_1,t_2]$, then given that its preconditions $\Gamma_{pre}(t_1,\bar{v})$ are satisfied, its effects, $\Gamma_{post}(t_1,t_2,\bar{v})$, will take place. Additionally, $\Gamma_{cons}(t_1,t_2,\bar{v})$ can be used to specify logical constraints associated with the action. For example, the following defines the elementary action **fly-to**: If a UAV should fly to a new position (x',y') within the temporal interval $[t,t']$, it must initially have sufficient fuel. At the next timepoint $t+1$ the UAV will not be hovering, and in the interval between the start and the end of the action, the UAV will arrive and its fuel level will decrease. Finally, there are two logical constraints bounding the possible duration of the flight action.

$$
\begin{aligned}
&[t,t'] \textbf{fly-to}(uav,x',y') \rightsquigarrow \quad [t]\ \mathsf{fuel}(uav) \geq \mathsf{fuel\text{-}usage}(uav, \mathsf{x}(uav), \mathsf{y}(uav), x', y') \to \\
&\quad R([t+1]\ \mathsf{hovering}(uav) \hat{=} False) \land R((t,t']\ \mathsf{x}(uav) \hat{=} x') \land R((t,t']\ \mathsf{y}(uav) \hat{=} y') \land \\
&\quad R((t,t']\ \mathsf{fuel}(uav) \hat{=} value(t, \mathsf{fuel}(uav) - \mathsf{fuel\text{-}usage}(uav, \mathsf{x}(uav), \mathsf{y}(uav), x', y'))) \land \\
&\quad t' - t \geq value(t, \mathsf{min\text{-}flight\text{-}time}(uav, \mathsf{x}(uav), \mathsf{y}(uav), x', y')) \land \\
&\quad t' - t \leq value(t, \mathsf{max\text{-}flight\text{-}time}(uav, \mathsf{x}(uav), \mathsf{y}(uav), x', y'))
\end{aligned}
$$

The translation function $Trans()$ translates $\mathscr{L}$(ND) expressions into $\mathscr{L}$(FL), a first-order logical language [5]. This provides a well-defined formal semantics for narratives in $\mathscr{L}$(ND).

The $\mathscr{L}$(FL) language is order-sorted, supporting both types and subtypes for features and values. This is also reflected in $\mathscr{L}$(ND), where one often assumes variable types are correlated to variable names – for example, uav_3 implicitly ranges over AVs. There are a number of sorts for values $\mathscr{V}_i$, including the Boolean sort $\mathscr{B}$ with the constants $\{true, false\}$. $\mathscr{V}$ is a supersort of all value sorts. There are a number of sorts for features $\mathscr{F}_i$, each one associated with a value sort $dom(\mathscr{F}_i) = \mathscr{V}_j$ for some j. The sort $\mathscr{F}$ is a supersort of all fluent sorts. There is also an action sort $\mathscr{A}$ and a temporal sort $\mathscr{T}$. Generally, t,t' will denote temporal variables, while $\tau, \tau', \tau_1, \ldots$ are temporal terms. $\mathscr{L}$(FL) currently uses the following predicates, from which formulas can be defined inductively using standard rules, connectives and quantifiers of first-order logic.

- $Holds\colon \mathscr{T} \times \mathscr{F} \times \mathscr{V}$, where $Holds(t,f,v)$ expresses that a feature f has a value v at a timepoint t, corresponding to $[t]\ f \hat{=} v$ in $\mathscr{L}$(ND).

- *Occlude*: $\mathscr{T} \times \mathscr{F}$, where $Occlude(t, f)$ expresses that a feature f is permitted to change values at time t. This is implicit in reassignment, $R([t]\ f \hat{=} v)$, in $\mathscr{L}$(ND).
- *Occurs*: $\mathscr{T} \times \mathscr{T} \times \mathscr{A}$, where $Occurs(t_s, t_e, A)$ expresses that a certain action A occurs during the interval $[t_s, t_e]$. This corresponds to $[t_s, t_e]A$ in $\mathscr{L}$(ND).

Trans() first generates the appropriate $\mathscr{L}$(FL) formulas corresponding to each $\mathscr{L}$(ND) statement. For example, the translation of the action specification above is as follows, where semantic attachment is used for greaterequals and minus and where *Occlude* formulas are generated by expanding the R macro.

$$\begin{array}{l}
\forall t, t', uav, x', y'[\\
Occurs(t, t', \textsf{\textbf{fly-to}}(uav, x', y')) \to (\\
\quad Holds(t, \textsf{greaterequal}(\textsf{fuel}(uav), \textsf{fuel-usage}(uav, \textsf{x}(uav), \textsf{y}(uav), x', y'))) \to\\
\quad Holds(t+1, \textsf{hovering}(uav), false) \wedge Holds(t', \textsf{x}(uav), x') \wedge Holds(t', \textsf{y}(uav), y') \wedge\\
\quad Holds(t', \textsf{fuel}(uav), value(t, \textsf{minus}(\textsf{fuel}(uav), \textsf{fuel-usage}(uav, \textsf{x}(uav), \textsf{y}(uav), x', y')))) \wedge\\
\quad t' - t \geq value(t, \textsf{min-flight-time}(uav, \textsf{x}(uav), \textsf{y}(uav), x', y')) \wedge\\
\quad t' - t \leq value(t, \textsf{max-flight-time}(uav, \textsf{x}(uav), \textsf{y}(uav), x', y')) \wedge\\
\quad \forall u[t < u \leq t' \to Occlude(u, \textsf{x}(uav)) \wedge Occlude(u, \textsf{y}(uav)) \wedge Occlude(u, \textsf{fuel}(uav))] \wedge\\
\quad Occlude(t+1, \textsf{hovering}(uav)))]
\end{array}$$

Foundational axioms such as unique names and domain closure axioms are appended when required. Logical entailment then allows the reasoner to determine when actions must occur. To ensure that they *cannot* occur at other times than explicitly stated, filtered circumscription is used. This also ensures that fluents can change values only when explicitly affected by an action or dependency constraint [5]. Pragmatically, this is done by representing possible change using the *Occlude* relation and then minimizing *Occlude* in a subset of the theory in question.

The structure of $\mathscr{L}$(ND) statements ensures that the second-order circumscription axioms are reducible to equivalent first-order formulas, allowing classical first-order theorem proving techniques to be used [5]. For unmanned systems, however, the logic will primarily be used to ensure a correct semantics for planners, execution monitors and mission specification languages and correlating this semantics closely to the implementation. Using TAL neither requires nor excludes theorem proving on board.

3.1 Task Planning using TALplanner

TALplanner [15, 16] is a domain-independent concurrent temporal planner where the declarative semantics for planning domains and problem instances is directly based on TAL. Inspired by TLplan [2], it also supports the use of *domain-specific control formulas* in TAL. Such formulas act as requirements on the set of valid solutions: A plan is a *solution* only if its final state satisfies the goal *and* all control formulas are satisfied in the complete state sequence that would result from executing the plan, which can be viewed as a logical model. This serves two separate purposes. First, it allows the specification of complex temporally extended goals such as safety conditions that must be upheld throughout the (predicted) execution of a plan. Second, the additional constraints on the final solution often allow the planner to prune entire branches of the search tree – whenever it can be proven that every search node on the branch corresponds to a state sequence that violates at least one control rule.

As an example, consider two simple control rules that could be used in an airplane-based logistics domain. First, a package should only be loaded onto a plane if a plane is required to move it: If the goal requires it to be at a location in another city. Regardless of which operator is used to load a package, one can detect this through the fact that it is in a plane at time $t+1$, but was *not* in the same plane at time t.

$$\begin{array}{l}
\forall t, obj, plane, loc.[t] \neg \textsf{in}(obj, plane) \wedge \textsf{at}(obj, loc) \wedge \textsf{[t+1]}\ \textsf{in}(obj, plane) \to\\
\qquad\qquad \exists loc'\ [\ \textsf{goal}\ (\textsf{at}(obj, loc')) \wedge \textsf{[t]}\ \textsf{city\_of}(loc) \not\hat{=} \textsf{city\_of}(loc')\ ]
\end{array}$$

Second, if a package is at its destination, it should not be moved.

$$\forall t, obj, loc.[t]\mathsf{at}(obj, loc) \wedge \mathsf{goal}\ (\mathsf{at}(obj,\ loc)) \rightarrow \mathsf{[t+1]}\ \mathsf{at}(obj,\ loc)$$

Surprisingly, such simple hints to an automated planner can often improve planning performance by orders of magnitude given that the planner has the capability to make use of the hints.

Related work. Though a large variety of planners exist in the literature, most lack the ability to take advantage of additional domain knowledge. Of those who do, most are based on Hierarchical Task Networks [17–19], where the underlying idea is that every objective that one may like to achieve is associated with a means of reducing it to more primitive objectives, until elementary actions are reached. Compared to this approach, the use of domain-specific control formulas allows domain knowledge to be added in a more incremental fashion while the planner still takes care of a larger proportion of the work required to find a plan.

3.2 Execution Monitoring

The execution monitoring system described here is based on an intuition similar to the one underlying the temporal control formulas used in TALplanner. As a plan is being executed, information about the surrounding environment is sampled at a given frequency by DyKnow (section 2). Each new sampling point generates a new state which provides information about all state variables used by the current monitor formulas, thereby providing information about the *actual* state of the world as opposed to what could be *predicted* from the domain specification. The resulting sequence of states corresponds to a partial logical interpretation, where "past" and "present" states are completely specified whereas "future" states are completely undefined.

Monitor formulas are expressed in a variation of TAL augmented with a set of *tense operators* similar to those used in modal tense logics such as MTL [14]. This allows the expression of complex metric temporal conditions and is amenable to incremental evaluation as each new state is generated. Violations can then be detected as early and as efficiently as possible using a *formula progression* algorithm, while the basis in TAL provides a common formal semantic ground for planning and monitoring.

Three tense operators have been introduced into $\mathscr{L}$(ND): U (until), $\Diamond$ (eventually), and $\Box$ (always). Like all $\mathscr{L}$(ND) expressions, these operators are macros on top of the first order base language $\mathscr{L}$(FL).

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

- $\tau \leq \tau'$, $\tau < \tau'$, *or* $\tau = \tau'$, *where* τ *and* τ' *are temporal terms,*
- $\omega \leq \omega'$, $\omega < \omega'$, *or* $\omega = \omega'$, *where* ω *and* ω' *are value terms,*
- *f, where f is a boolean fluent term (state variable term),*
- $f \hat{=} \omega$*, where f is a fluent term and* ω *is a value term of the corresponding sort,*
- $\phi\ \mathsf{U}_{[\tau,\tau']}\ \psi$*, where* ϕ *and* ψ *are monitor formulas and* τ *and* τ' *are temporal terms,*
- $\Diamond_{[\tau,\tau']}\ \phi$*, where* ϕ *is a monitor formula and* τ *and* τ' *are temporal terms,*
- $\Box_{[\tau,\tau']}\ \phi$*, where* ϕ *is a monitor formula and* τ *and* τ' *are temporal terms, or*
- *a combination of monitor formulas using standard logical connectives and quantifiers.*

The shorthand notation $\phi\ \mathsf{U}\psi \equiv \phi\ \mathsf{U}_{[0,\infty)}\ \psi$*,* $\Diamond\ \phi \equiv \Diamond_{[0,\infty)}\ \phi$*, and* $\Box\ \phi \equiv \Box_{[0,\infty)}\ \phi$ *is also permitted.*

Tense operators use relative time, where each formula is evaluated relative to a "current" timepoint. The semantics of these formulas satisfies the following conditions (see [6] for details):

- $\phi\ \mathsf{U}_{[\tau,\tau']}\ \psi$ ("until") holds at time t iff ψ holds at some state with time $t' \in [t+\tau, t+\tau']$ and ϕ holds until then (at all states in $[t,t')$, which may be an empty interval).

- $\Diamond_{[\tau,\tau']}\,\phi$ ("eventually") is equivalent to $\texttt{true}\,\mathsf{U}_{[\tau,\tau']}\,\phi$ and holds at t iff ϕ holds in some state with time $t' \in [t+\tau, t+\tau']$.
- $\Box_{[\tau,\tau']}\,\phi$ is equivalent to $\neg\Diamond_{[\tau,\tau']}\neg\phi$ and holds at t iff ϕ holds in all states at time $t' \in [t+\tau, t+\tau']$.

Example 1. *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 "global" (always active) monitor formula can be used to detect violations of this specification:*

$$\Box\,\forall uav.(\mathsf{power}(uav) > M \rightarrow \mathsf{power}(uav) < f \cdot M\; \mathsf{U}_{[0,\tau]}\; \Box_{[0,\tau']}\,\mathsf{power}(uav) \leq M) \qquad \Box$$

Note that this does not *cause* the UAV to behave in the desired manner. The monitor formula instead serves as a method for detecting the failure of the helicopter control software to function as required.

Example 2. *The following monitor formula can be attached directly to the* pickup-box *action. Each time a* pickup-box *action is executed, the formula will be instantiated with the uav and box parameters and evaluation will begin.*

$$\Diamond_{[0,5000]}\,\Box_{[0,1000]}\,\mathit{carrying}(uav, box)$$

Within 5000 ms, the UAV should detect that it is carrying the box, and it should detect this for at least 1000 ms. The latter condition protects against problems during the pickup phase, where the box may be detected during a very short period of time even though the ultimate result is failure. □

To promptly detect violations of monitor conditions during execution, a *formula progression* algorithm is used [1]. By definition, a formula ϕ holds in the state sequence $[s_0, s_1, \ldots, s_n]$ iff $\mathsf{Progress}(\phi, s_0)$ holds in $[s_1, \ldots, s_n]$. In essence, this evaluates those parts of the monitor formula that refer to s_0, returning a new formula to be progressed in the same manner once s_1 arrives.

If the formula $\bot$ (false) is returned, then sufficient information has been received to determine that the monitor formula must be violated regardless of the future development of the world. For example, this will happen as soon as the formula $\Box\,\mathsf{speed} < 50$ is progressed through a state where $\mathsf{speed} \geq 50$. This signals a potential or actual failure from which the system must attempt to *recover*. Recovery procedures can be associated with specific monitor formulas and operators. For example, if a UAV fails to take off with a certain cargo, it can adjust its assumptions about how much it is able to lift. This feeds back information from the failure into the information given to the planner for replanning.

If $\top$ (true) is returned, the formula must instead hold regardless of what happens "in the future". In other cases, the state sequence complies with the constraint "so far", and progression returns a new and potentially modified formula that should be progressed again as soon as another state is available.

Definition 2 (Progression of Monitor Formulas)**.** *The following algorithm is used for progression of monitor formulas. States are not first-class objects in* TAL *and are therefore identified by an interpretation $\mathscr{I}$ corresponding to an entire state sequence and a timepoint τ identifying a state within that sequence. Special cases for $\Box$ and $\Diamond$ can be introduced for performance.*

```
procedure Progress($\phi, \tau, \mathscr{I}$)
if $\phi = f(\bar{x}) \hat{=} v$
  if $\mathscr{I} \models Trans\big([\tau]\ \phi\big)$ return $\top$ else return $\bot$
if $\phi = \neg\phi_1$ return $\neg\mathsf{Progress}(\phi_1, \tau, \mathscr{I})$
if $\phi = \phi_1 \otimes \phi_2$ return $\mathsf{Progress}(\phi_1, \tau, \mathscr{I}) \otimes \mathsf{Progress}(\phi_2, \tau, \mathscr{I})$
if $\phi = \forall x.\phi$ return $\bigwedge_{c \in X} \mathsf{Progress}(\phi[x \mapsto c], \tau, \mathscr{I})$ // where $x$ is a variable of sort $X$
if $\phi = \exists x.\phi$ return $\bigvee_{c \in X} \mathsf{Progress}(\phi[x \mapsto c], \tau, \mathscr{I})$ // where $x$ is a variable of sort $X$
if $\phi$ contains no tense operator
  if $\mathscr{I} \models Trans\big(\phi\big)$ return $\top$ else return $\bot$
if $\phi = \phi_1 \mathsf{U}_{[\tau_1,\tau_2]} \phi_2$
  if $\tau_2 < 0$ return $\bot$
  elsif $0 \in [\tau_1,\tau_2]$ return $\mathsf{Progress}(\phi_2,\tau,\mathscr{I}) \vee (\mathsf{Progress}(\phi_1,\tau,\mathscr{I}) \wedge (\phi_1 \mathsf{U}_{[\tau_1-1,\tau_2-1]} \phi_2))$
  else return $\mathsf{Progress}(\phi_1,\tau,\mathscr{I}) \wedge (\phi_1 \mathsf{U}_{[\tau_1-1,\tau_2-1]} \phi_2)$
```

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

Empirical testing has been performed using common formula patterns exercising complex combinations of time and modality, together with synthetic state inputs designed to exercise both the best and the worst cases for these formulas. An example is $\Box(\mathsf{speed}(uav) > T \to \Diamond_{[0,1000]}\,\Box_{[0,1000]}\,\mathsf{speed}(uav) \leq T)$: If the UAV is flying too quickly, then within 1000 ms, there must begin a period lasting at least 1000 ms where it is within the limits. Even with the worst case input and with new states arriving every 100 ms, 1500 formulas of this form could be progressed in parallel even using an on-board UAV computer with a comparatively old 1.4 GHz Pentium M CPU. Similar results have been shown for formulas of different forms, indicating the general feasibility of this approach even with limited computational power [6].

Related Work. The most common approach to execution monitoring uses a predictive model to determine what state a robot should be in, continuously comparing this to the current state as detected by sensors [3, 9, 21]. However, the fact that one can detect a discrepancy between the current state and the predicted state does not necessarily mean that this discrepancy has a detrimental effect on execution or mission achievement. Thus, one must take great care to distinguish essential deviations from unimportant ones. Using explicit monitor formulas allows us to explicitly distinguish what is *necessary* from what is merely *predicted*.

4 Conclusions

As Israel [13] points out, logic has played two very broad roles in AI:

(i) as a source of languages and logics for artificial reasoners;

(ii) most importantly, as a source of analytical tools and techniques; more broadly, as providing the underlying mathematical and conceptual framework within which much of AI research is done. (p. 2, [13])

As demonstrated in this paper, both these roles are fundamentally important in the development of highly autonomous robotic systems. Due to the complexity of such systems, one requires a principled means of controlling that complexity. The use of temporal logic as an analytical tool to provide the underlying conceptual framework for many of the functionalities that make up an autonomous robotic system is highly beneficial in this respect. The three functionalities described in this paper are formally grounded in this manner. Since one requires soft real-time reasoning capabilities, one has to be somewhat more creative in the use of logic as the basis for a reasoning component. The use of stream-based reasoning techniques provides a novel use of logic as an online reasoner. The autonomous system has the capability of dynamically generating temporal models and then querying these models in soft realtime. This capability has been used with great success as a basis for efficient planning, execution monitoring, diagnosis and as a constraint checker for safety and liveness conditions.

References

[1] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22, 1998.

[2] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:123–191, 2000.

[3] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau. Using iterative repair to improve the responsiveness of planning and scheduling. In *Proc. 5th International Conference on Artificial Intelligence Planning Systems (AIPS-2000)*, pages 300–307, Breckenridge, Colorado, USA, April 2000. AAAI Press.

[4] P. Doherty. Advanced research with autonomous unmanned aerial vehicles. In *Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2004. Extended abstract for plenary talk.

[5] P. Doherty and J. Kvarnström. Temporal action logics. In *The Handbook of Knowledge Representation*, chapter 18, pages 709–757. Elsevier, 2008.

[6] P. Doherty, J. Kvarnström, and F. Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *J. of Autonomous Agents and Multi-Agent Systems*, 19(3):332–377, 2009.

[7] P. Doherty, J. Kvarnström, M. Wzorek, P. Rudol, F. Heintz, and G. Conte. HDRC3: A distributed hybrid architecture for unmanned aircraft systems. In *Handbook of Unmanned Aerial Vehicles (forthcoming)*. Springer, 2014.

[8] Patrick Doherty, Joakim Gustafsson, Lars Karlsson, and Jonas Kvarnström. (TAL) temporal action logics: Language specification and tutorial. *Electronic Transactions on Artifical Intelligence*, 2(3-4):273–306, 1998.

[9] R. Fikes. Monitored execution of robot plans produced by STRIPS. In *Proc. IFIP Congress (IFIP-1971)*, pages 189–194, Ljubljana, Yugoslavia, 1971.

[10] F. Heintz. *DyKnow: A Stream-Based Knowledge Processing Middleware Framework*. PhD thesis, Department of Computer and Information Science, Linköping University, 2009.

[11] F. Heintz and P. Doherty. DyKnow: An approach to middleware for knowledge processing. *J. of Intelligent and Fuzzy Systems*, 15(1):3–13, 2004.

[12] F. Heintz, J. Kvarnström, and P. Doherty. Stream-based hierarchical anchoring. *Künstliche Intelligenz*, 27:119–128, 2013.

[13] David J Israel. The role(s) of logic in artificial intelligence. *Handbook of Logic in Artificial Intelligence and Logic Programming*, 1:1–29, 1993.

[14] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4), 1990.

[15] J. Kvarnström. *TALplanner and Other Extensions to Temporal Action Logic*. PhD thesis, Department of Computer and Information Science, Linköping University, 2005.

[16] Jonas Kvarnström and Patrick Doherty. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30:119–169, 2000.

[17] Dana S. Nau, T. C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wo, and F. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404, December 2003.

[18] Earl D. Sacerdoti. The nonlinear nature of plans. In *Proc. Fourth International Joint Conference on Artificial Intelligence (IJCAI-1975)*, pages 206–214, Tiblisi, Georgia, USSR, 1975.

[19] Austin Tate. Generating project networks. In *Proc. Fifth International Joint Conference on Artificial Intelligence (IJCAI-1977)*, pages 888–893, Cambridge, Massachusetts, USA, August 1977.

[20] Moritz Tenorth and Michael Beetz. KnowRob – A Knowledge Processing Infrastructure for Cognition-enabled Robots. Part 1: The KnowRob System. *Int. J. of Robotics Research*, 32(5), 2013.

[21] R. Washington, K. Golden, and J. Bresina. Plan execution, monitoring, and adaptation for planetary rovers. *Electronic Transactions on Artificial Intelligence*, 5(17), 2000.