Allen Linear (Interval) Temporal Logic –Translation to LTL and Monitor Synthesis–

Grigore Roşu[1] * and Saddek Bensalem[2]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign, USA
[2] VERIMAG, 2 Avenue de Vignate, 38610 Gieres, France

Abstract. The relationship between two well established formalisms for temporal reasoning is first investigated, namely between Allen's interval algebra (or Allen's temporal logic, abbreviated ATL) and linear temporal logic (LTL). A discrete variant of ATL is defined, called Allen linear temporal logic (ALTL), whose models are ω-sequences of timepoints, like in LTL. It is shown that any ALTL formula can be linearly translated into an equivalent LTL formula, thus enabling the use of LTL techniques and tools when requirements are expressed in ALTL. Then the monitoring problem for ALTL is discussed, showing that it is NP-complete despite the fact that the similar problem for LTL is EXPSPACE-complete. An effective monitoring algorithm for ALTL is given, which has been implemented and experimented with in the context of planning applications.

1 Introduction

Allen's interval algebra, also called Allen's temporal logic (ATL) in this paper, is one of the best established formalisms for temporal reasoning [9]. It is frequently used in AI, especially in planning. Linear temporal logic (LTL) [13] is successfully applied in program verification, temporal databases, and related domains. Despite the widespread use of both ATL and LTL, there is no formal and systematic investigation of their relationship. This paper makes a step in this direction. To have a semantic basis for such a relationship, we define a discrete variant of ATL, called Allen linear temporal logic (ALTL), whose syntax and complexity of satisfiability are the same as in ATL, but whose models resemble those of LTL.

We show that ALTL can be linearly encoded into a subset of LTL. This encoding yields the NP-completeness of the satisfiability problem for an ATL (proposed in [5]) slightly richer than the original one proposed by Allen. On the practical side, this result allows us to use the plethora of techniques and analysis tools developed for LTL on requirements (or compatibilities) expressed using ATL. Since ATL is *the* logic of planning, and since validation and verification (V&V) of complex plans for systems with decisional autonomy is highly desirable, if not crucial, in many applications, this automated translation into LTL potentially enables us to use well-understood V&V techniques and tools in a domain lacking (but in need of) them. Further, it may also support the suggestion made in [3]

* Supported by NSF grants CCF-0234524, CCF-0448501, and CNS-0509321.

that LTL can be itself seriously regarded as a suitable formalism for temporal reasoning in AI, and particularly in planning. There are, however, complexity aspects that cannot be ignored (some of them pointed in this paper).

The importance of monitoring in planing cannot be overestimated. For example, an autonomous rover whose execution plans have been rigorously verified may still fail for reasons such as hardware or operating system failures, unexpected terrain in an unknown environment, etc. Having monitors to check online the execution of plans step by step and to trigger recovery code in case of violations is of crucial importance. It is the challenge of generating efficient monitors from planning requirements that motivated the work in this paper.

We argue that a blind use of monitoring algorithms for LTL to monitor ALTL formulae is not feasible even on small ALTL formulae, and then give a special-purpose monitoring algorithm for ALTL which only needs to call a boolean satisfiability checker at each step, on a boolean proposition smaller in size than the original ALTL-formula. This algorithm also proves that the monitoring problem for ALTL is NP-complete, in spite of the fact that the monitoring problem for LTL is EXPSPACE-complete. This monitoring algorithm has been implemented and experimented with in the context of planning for autonomous rovers.

Preliminaries. We assume the reader familiar with Linear Temporal Logic (LTL) [13]. We here only recall some basics and introduce our notation. LTL is interpreted in "flows of time", modeled as strict linear orders $(T, <)$, where T is a nonempty set of "time points".The LTL language consists of propositional symbols (p_0, p_1, $\cdots$), boolean operators ($\neg$ and $\wedge$), and temporal operators $\mathcal{U}$ ("until") and $\circ$ ("next"), and LTL formulae follow the common syntax

$$\varphi ::= p \ \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \,\mathcal{U}\, \varphi_2 \mid \ \circ \varphi.$$

LTL models are triples $M = (T, <, v)$ such that $(T, <)$ is a strict total order (a flow of time) and, v is a map called valuation associating with each variable p a set $v(p) \subseteq T$ of time points (where p is supposed to be true). The satisfaction relation $M \models \varphi$ is defined as in [13].

Other important temporal operators, such as $\Diamond$(eventually) and $\Box$ (always), are expressible using $\mathcal{U}$, such as $\Diamond\varphi =$ true $\mathcal{U}\ \varphi$ (φ will eventually hold) and $\Box\varphi = \neg\Diamond\neg\varphi$ (φ will always hold). The operator $\Diamond$ can also be expressed in terms of $\Box$, namely $\Diamond\varphi = \neg\Box\neg\varphi$. In this paper we only need the fragment of LTL with $\Diamond$ and $\Box$, without $\circ$ and $\mathcal{U}$. Since $\Diamond$ and $\Box$ can be defined in terms of each other, we take the liberty to call this fragment $\mathsf{LTL}_\Box$ (we could have also called it $\mathsf{LTL}_\Diamond$). The "satisfiability problem" for a formula φ is concerned with whether there is some model M such that $M \models \varphi$. The satisfiability problem of LTL formulae is PSPACE-complete, while the satisfiability of $\mathsf{LTL}_\Box$ is NP-complete [15].

2 Allen (Linear) Temporal Logic - ATL (ALTL)

Allen Temporal Logic *(ATL)* [1] is specified as a framework to deal with incomplete relative temporal information such as "Event A is before event B".

Instead of adopting time points, Allen takes intervals as the primitive temporal quantity. He introduced 13 (mutually exclusive) basic binary relations between any two intervals. In AI planning, ATL is used to reason about concurrency and temporal extent; action instances and states are described in terms of temporal intervals that are linked by constraints. Attributes whose states change over time are called *state variables* and can be thought of as concurrent threads. The history of values of a state variable over a period of time is called a "timeline". Thus, a timeline consists of a sequence of intervals. It is convenient to think of each interval as a structure unit, called *token* and involving a corresponding procedure. The interval constraints among all possible values that must occur among tokens for a plan to be legal are organized in a set of *compatibilities.* A compatibility determines the necessary correlation with other procedure invocations in a legal plan, i.e., which procedures must precede or follow others, which should be co-temporal, etc.

Example 1. To illustrate the use of ATL, let us consider the classical (in panning) monkey/banana problem, used as a running example. A monkey is at location "x", the banana is hanging from the tree. The monkey is at height "Low", but if it climbs the tree then it will be at height "High", same as the banana. The available actions are "Going" from a place to another, "Climbing" (up) and "Climbing Down", and "Grabbing" an object.

Attributes. *BANANA* has one state variable "*Banana-sv*" saying if the monkey has the banana or not. *LOCATION* has one variable "*Location-sv*" for the location of the monkey. *ALTITUDE* has one variable "*Altitude-sv*" for the height.

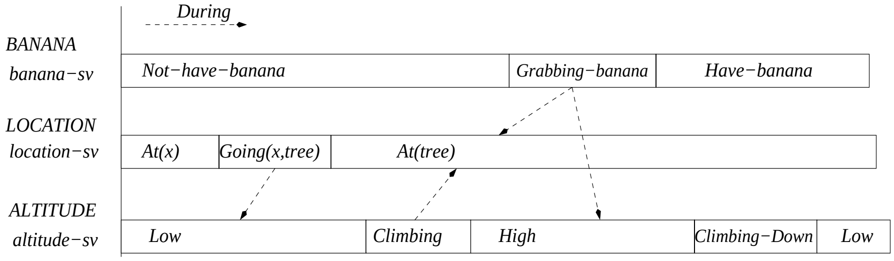


Fig. 1. Attributes and compatibilities

Compatibilities. Now that we have the state variables, we can consider their *compatibilities* (i.e., causal and temporal relationships between attributes):

- *Have-banana* ("H_b") requires *Grabbing-banana* ("G_b") which requires *Not-have-banana* ("N_{hb}"). *Grabbing-banana* is performed while *High* and *At(tree).*
- *At(tree)* ("@$(tree)$") requires going from the location *"x"* to the tree which requires *At(x)* ("@(x)"). *Going(x,tree)* ("G(x,tree)") is performed while *Low.*
- *High* ("H") requires *Climbing* ("C") which requires *Low* ("L"), and *Climbing-Down* ("C_D") requires *High. Climbing* is performed while *At(tree).*

These compatibilities can be formally specified in ATL as follows:

$$\begin{array}{l} \mathit{Meets}(N_{hb}, G_b) \wedge \mathit{Meets}(G_b, H_b) \wedge \mathit{During}(G_b, @(\mathit{tree})) \wedge \mathit{During}(G_b, H)) \wedge \\ \mathit{Meets}(@(x), G(x, \mathit{tree})) \wedge \mathit{Meets}(G(x, \mathit{tree}), @(\mathit{tree})) \wedge \mathit{During}(G(x, \mathit{tree}), L) \wedge \\ \mathit{Meets}(L, C) \wedge \mathit{Meets}(C, H) \wedge \mathit{Meets}(H, C_D) \wedge \mathit{Meets}(C_D, L) \wedge \mathit{During}(C, @(\mathit{tree})). \end{array}$$

Let us consider the subformula consisting of the first four conjuncts above, and suppose that an unexpected "flying monkey" wants the banana. It climbs the tree, but it cannot reach for the banana. Being a flying monkey, it jumps for the banana, grabs it while gliding when it is still *High* and *At(tree)*, but as it glides it leaves the tree location. Supposing that it leaves the tree location at the same time it changes the status from *Grabbing-banana* to *Have-banana*, one can notice that the third conjunct is violated. Indeed, G_b must hold *during* $@(\mathit{tree})$, meaning that there must be some (non-zero) periods of time in which the monkey was at the tree location before and after grabbing the banana.

It is often useful to state that some propositions hold all the time or eventually during an interval. For example, assume one more state predicate, *hungry*, saying whether the monkey is hungry or not, and assume that we want to state that monkeys should grab and have bananas only if they are hungry and do not already have bananas. This can be done with the following additional conjunct:

$\mathit{Occurs}(\mathit{hungry}, N_{hb}) \wedge \mathit{Holds}(\mathit{hungry}, G_b) \wedge \mathit{Holds}(\mathit{hungry}, H_b)$ □

There are different views on how intervals should be modeled in different time flows. A common interpretation is that the intervals are ordered pairs of distinct points in $\mathbb{Q}$ or $\mathbb{R}$. For simplicity, it is convenient to use semantics where intervals are arbitrary convex non-empty subsets of time points of an arbitrary time flow.

Definition 1. *If $\mathcal{P}$ is a set of* ***atomic propositions*** *and $\mathcal{I}$ is a set of* ***intervals****, then an* ***Allen temporal logic formula formula over $\mathcal{P}$ and*** *$\mathcal{I}$, or an ATL($\mathcal{P}$,$\mathcal{I}$)-**formula** or even just a* ***formula*** *when $\mathcal{P}$ and $\mathcal{I}$ are understood from context, is any boolean combination of* ***basic formulae*** *of the form:*

- *$\mathit{Equals}(i, j)$,*
- *$\mathit{Before}(i, j)$ and $\mathit{After}(i, j)$,*
- *$\mathit{Overlaps}(i, j)$ and $\mathit{OverlappedBy}(i, j)$,*
- *$\mathit{Meets}(i, j)$ and $\mathit{MetBy}(i, j)$,*
- *$\mathit{Contains}(i, j)$ and $\mathit{During}(i, j)$,*
- *$\mathit{Starts}(i, j)$ and $\mathit{StartedBy}(i, j)$,*
- *$\mathit{Ends}(i, j)$ and $\mathit{EndedBy}(i, j)$,*
- *$\mathit{Holds}(p, i)$ and $\mathit{Occurs}(p, i)$,*

where $i, j \in \mathcal{I}$ and $p \in \mathit{Bool}(\mathcal{P})$.

$\mathit{Bool}(\mathcal{P})$ is the set of boolean propositions over variables in $\mathcal{P}$. Interestingly, the original formulation of ATL [1] did not include *Holds* and *Occurs*; motivated by practical reasons, they were added later in [5]. To define a formal semantics of ATL we need to first define an appropriate notion of model.

Definition 2. *Let $(T, <)$ be a strict total order, i.e., $<$ is transitive and anti-symmetric (not reflexive). The relation $<$ is tacitly extended to a strict partial order on subsets of T, namely $X < Y$ iff $x < y$ for all $x \in X$ and $y \in Y$. Also, by abuse of notation, we may write just x instead $\{x\}$; thus, $x < Y$ means that $x < y$ for all $y \in Y$. For $x, y \in T$ let (x, y) be the set $\{z \in T \mid x < z < y\}$. A subset C of T is* ***<-convex****, or simply* ***convex****, iff $(x, y) \subseteq C$ for any $x, y \in C$.*

In $\mathbb{R}$, for example, the convex sets are precisely the intervals. Recall that intervals in $\mathbb{R}$ can be open or closed on any of their ends, and that they may be bound by $-\infty$ or $+\infty$ at their left or right ends, respectively.

Definition 3. *A* ***$(\mathcal{P},\mathcal{I})$-interval model*** *(or simply an* ***interval model*** *when $\mathcal{P}$ and $\mathcal{I}$ are understood) is a structure $\mathcal{M} = (T, <, v, \sigma)$, where $(T, <)$ is a strict total order (modeling the intended flow of time), $v : \mathcal{P} \to 2^T$ is a valuation map assigning to each atomic proposition $p \in \mathcal{P}$ a set of time points $v(p)$ (in which the proposition is assumed to be true), and σ is a map that associates with every interval $i \in \mathcal{I}$ a non-empty convex subset $\sigma(i)$ of T. We may also refer to $(\mathcal{P},\mathcal{I})$-interval models as* ***models of*** *$\mathsf{ATL}(\mathcal{P},\mathcal{I})$.*

We are now ready to give the semantics of ATL.

Definition 4. *An interval model $\mathcal{M} = (T, <, v, \sigma)$* ***satisfies****:*

- *Equals(i, j) iff $\sigma(i) = \sigma(j)$;*
- *Before(i, j) or After(j, i) iff there is some $t \in T$ such that $\sigma(i) < t < \sigma(j)$;*
- *Overlaps(i, j) or OverlappedBy(j, i) iff $\sigma(i) \cap \sigma(j) \neq \emptyset$ and there are some $t_i \in \sigma(i)$ and $t_j \in \sigma(j)$ such that $t_i < \sigma(j)$ and $\sigma(i) < t_j$;*
- *Meets(i, j) or MetBy(j, i) iff $\sigma(i) < \sigma(j)$ and there is no $t \in T$ such that $\sigma(i) < t < \sigma(j)$;*
- *Contains(i, j) or During(j, i) iff there are some $t_i, t'_i \in \sigma(i)$ such that $t_i < \sigma(j) < t'_i$;*
- *Starts(i, j) or StartedBy(j, i) iff $\sigma(i) \subset \sigma(j)$, there is no $t_j \in \sigma(j)$ such that $t_j < \sigma(i)$, but there is some $t_j \in \sigma(j)$ such that $\sigma(i) < t_j$;*
- *Ends(i, j) or EndedBy(j, i) iff $\sigma(i) \subset \sigma(j)$, there is no $t_j \in \sigma(j)$ such that $\sigma(i) < t_j$, but there is some $t_j \in \sigma(j)$ such that $t_j < \sigma(i)$;*
- *Holds(p, i) iff $\sigma(i) \subseteq v(p)$; and*
- *Occurs(p, i) iff $\sigma(i) \cap v(p) \neq \emptyset$ iff $\neg$Holds$(\neg p, i)$.*

Satisfaction is defined as usual on boolean combinations of ATL formulae. We use the notation $\mathcal{M} \models_{\mathsf{ATL}} \varphi$ to denote the fact that the interval structure $\mathcal{M}$ satisfies the ATL formula φ.

Therefore, *Meets*(i, j) holds iff j starts *immediately* after i. On the other hand, *Before*(i, j) holds iff i starts and ends before j, but there is also some proper time elapsed between the end of i and the beginning of j. *Overlaps*(i, j) holds iff i starts strictly before j starts, they have some common time points, and i ends strictly before j ends. *Contains*(i, j) holds iff j starts strictly after i starts and terminates strictly before i terminates. *Starts*(i, j) holds iff i and j start together but j continues (strictly) after i ends; dually, $\mathit{Ends}(i,j)$ holds iff i and j terminate together but j starts strictly before i starts. $\mathit{Holds}(p,i)$ is satisfied iff p holds at any time point in i, while $\mathit{Occurs}(p,i)$ is satisfied iff p holds at some time point in i. The NP-completeness of the satisfiability problem problem for ATL without *Holds* [16] gives us immediately the NP-hardness of our ATL with *Holds* above. We will show in the next section that it is NP-complete.

In many practical applications of interest, time elapses at a discrete and enumerable rate. We next define a variant of Allen temporal algebra in which the support of the interval models are ω-sequences of time points, that is, linear (infinite) sequences $t_1 < t_2 < t_3 < \cdots < t_n < \cdots$. We write these strict total orders compactly as $t_1t_2t_3 \ldots t_n \ldots$. We call the new logic ***Allen Linear Temporal Logic*** (ALTL). Note that ALTL has *the same syntax* as ATL and its satisfaction relation is defined like in ATL, but that its models are structures of the form $\mathcal{M} = (t_1t_2 \ldots, v, \sigma)$, where $t_1t_2 \ldots$ are ω-sequences of time points and σ maps intervals in $\mathcal{I}$ into non-empty convex sets $\sigma(i)$ of $T = \{t_1, t_2, \ldots\}$ (with the expected strict total ordering $<$ defined as $t_m < t_n$ iff $m < n$). It is easy to see that the convex sets of T are either finite sets of the form $\{t_m, t_{m+1}, \ldots, t_n\}$ for some $0 < m \leq n$, or infinite sets of the form $\{t_m, t_{m+1}, \ldots\}$ for some $0 < m$.

It is easy to see that the restricted models of ALTL do not affect in any way the complexity or expressivity of ATL. Indeed, for any model $\mathcal{M} = (T, <, v, \sigma)$ of ATL we can construct a model $\mathcal{M}' = (T', <', v', \sigma')$ of ALTL as follows:

1. Define on T the "behavioral" equivalence relation $\sim$ with respect to interval and atomic proposition memberships; formally, $t_1 \sim t_2$ iff (for any atomic proposition p, either $t_1, t_2 \in v(p)$ or $t_1, t_2 \notin v(p)$) and (for any interval i, either $t_1, t_2 \in \sigma(i)$ or $t_1, t_2 \notin \sigma(i)$);
2. Add to T' precisely one element from each equivalence class of $\sim$; therefore, $T' \subseteq T$;
3. take $<'$, v' and σ' to be the restrictions of $<$, v and σ to T', respectively.

It is easy to see that $\mathcal{M}'$ is a model of ALTL which satisfies precisely the same formulae that $\mathcal{M}$ satisfies in ATL (syntax is the same in ATL and ALTL), because they satisfy the same basic formulae.

Supposing that $\mathcal{M}$ is the continuous behavior of a dynamic system that one wants to observe or monitor, in order to ensure the correctness of the monitoring process (i.e., that ATL-requirements violations are reported correctly at runtime) one should make sure that the observation points are frequent enough to guarantee that at least one element (timepoint or state snapshot) of each equivalence class has been observed by the monitor. If this property is not respected then erroneous behaviors can be wrongly reported or missed. For example, a "meets" (or "before") relation may look correct (wrong) just because no timepoint was generated in-between the two intervals, so they look as if they meet each other. The problem of efficient *system instrumentation* to emit a minimal, but not less than necessary, number of events to system observers/monitors is certainly very important (to reduce the undesirable runtime monitoring overhead) and seems interesting, but we do not discuss it here. From here on we assume that models of ALTL are available.

3 Linear Translation of ALTL into LTL

We next define an automatic encoding of ALTL into $\mathsf{LTL}_{\Box}$. Encoding a logic into another logic is a technically intricate concept, which can be defined quite precisely but which we avoid discussing here. Instead, we here define our ALTL-2-LTL encoding at syntactic level and then just state and prove its semantic correctness. The reader interested in a deeper understanding of why the results below indeed give an encoding of one logic into another logic is referred to [6].

Note that the models of ALTL differ from those of LTL in that they contain a concrete interpretation for each interval. Therefore, in order to establish a semantic relationship between the models of the two logics, we need to first add syntactic support for "intervals" to LTL. A simple way to do this is to add an atomic propositional symbol $\in_i$ to the syntax of LTL for each interval $i \in \mathcal{I}$, with the intuition that a time point is in the interval i in a model of ALTL if and only if the proposition $\in_i$ holds in that time point in the corresponding model of LTL. Moreover, we need to also capture via corresponding LTL formulae the fact that intervals are interpreted into non-empty convex sets in ALTL models.

Definition 5. *Let $\mathcal{P}_{\mathcal{I}}$ be the set of atomic propositions $\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}\}$ and let $\Psi_{\mathcal{I}}$ be the set of LTL formulae $\{\psi_i \mid i \in \mathcal{I}\}$ over propositions in $\mathcal{P}_{\mathcal{I}}$, where ψ_i is the formula $\Diamond{\in_i} \wedge \neg\Diamond(\in_i \wedge \Diamond(\neg{\in_i} \wedge \Diamond{\in_i}))$ for each $i \in \mathcal{I}$.*

The following establishes the relationship between models of ALTL and of LTL:

Proposition 1. *There is a bijection between $(\mathcal{P}, \mathcal{I})$-interval models and models of $\mathsf{LTL}(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}\})$ that satisfy $\Psi_{\mathcal{I}}$.*

Proof. Let $\mathcal{M} = (T, <, v, \sigma)$ be a tuple where $(T, <)$ is an ω-sequence, v is a map $\mathcal{P} \to 2^T$, and σ is a map $\mathcal{I} \to 2^T$; what $\mathcal{M}$ is missing to be a model of $\mathsf{ALTL}(\mathcal{P}, \mathcal{I})$ is the requirements that $\sigma(i)$ is non-empty and convex for any $i \in \mathcal{I}$. Then we can build a model $\mathcal{N} = (T, <, u)$ of $\mathsf{LTL}(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}\})$, where $u(p) = v(p)$ for all $p \in \mathcal{P}$ and $u(\in_i) = \sigma(i)$ for all $i \in I$. Conversely, for any model $\mathcal{N} = (T, <, u)$ of $\mathsf{LTL}(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}\})$ one can build a tuple $\mathcal{M} = (T, <, v, \sigma)$, where v is the restriction of u to $\mathcal{P}$ and $\sigma(i)$ is defined as $u(\in_i)$ for any $i \in \mathcal{I}$. What is left to prove is that $\sigma(i)$ is non-empty and convex for any $i \in \mathcal{I}$ if and only if $\mathcal{N} \models_{\mathsf{LTL}} \Psi_{\mathcal{I}}$. First, note that, for any $i \in \mathcal{I}$, $\sigma(i) \neq \emptyset$ is equivalent to $\mathcal{N} \models_{\mathsf{LTL}} \Diamond{\in_i}$. Second, since $\sigma(i)$ is convex if and only if there are no time points t_m, t_n, t_k with $0 < m < n < k$ such that $t_m, t_k \in \sigma(i)$ and $t_n \notin \sigma(i)$, one deduces that $\sigma(i)$ is convex if and only if $\mathcal{N} \models_{\mathsf{LTL}} \neg\Diamond(\in_i \wedge \Diamond(\neg{\in_i} \wedge \Diamond{\in_i}))$. Therefore, $\sigma(i)$ is non-empty and convex for each $i \in \mathcal{I}$ if and only if $\mathcal{N} \models_{\mathsf{LTL}} \Psi_{\mathcal{I}}$. □

Definition 6. *We let $[\cdot]$ define the bijection above, that is, if $\mathcal{M}$ is a $(\mathcal{P}, \mathcal{I})$-interval model then $[\mathcal{M}]$ is the corresponding model of $\mathsf{LTL}(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}\})$ satisfying $\Psi_{\mathcal{I}}$, defined as in the proof of Proposition 1.*

We are now ready to define the first part of our syntactic encoding of ALTL formulae into LTL formulae.

Definition 7. *Let* $[\cdot]$ *be the function taking formulae* φ *in* $\mathsf{ALTL}(\mathcal{P}, \mathcal{I})$ *into formulae* $[\varphi]$ *in* $\mathsf{LTL}(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}\})$ *defined inductively as follows:*

- $[\neg\varphi]$ *is* $\neg[\varphi]$;
- $[\varphi_1 \wedge \varphi_2]$ *is* $[\varphi_1] \wedge [\varphi_2]$;
- $[\mathit{Equals}(i, j)]$ *is* $\Box(\in_i \Leftrightarrow \in_j)$;
- $[\mathit{Before}(i, j)]$ *and* $[\mathit{After}(j, i)]$ *are* $\Diamond(\in_i \wedge \Diamond(\neg\in_i \wedge \neg\in_j \wedge \Diamond\in_j))$;
- $[\mathit{Meets}(i, j)]$ *and* $[\mathit{MetBy}(j, i)]$ *are*
  $\Diamond(\in_i \wedge \Diamond\in_j \wedge \neg\Diamond(\in_i \wedge \in_j) \wedge \neg\Diamond(\neg\in_i \wedge \neg\in_j \wedge \Diamond\in_j))$;
- $[\mathit{Overlaps}(i, j)]$ *and* $[\mathit{OverlappedBy}(j, i)]$ *are*
  $\Diamond(\in_i \wedge \neg\in_j \wedge \Diamond(\in_i \wedge \in_j \wedge \Diamond(\neg\in_i \wedge \in_j)))$;
- $[\mathit{Contains}(i, j)]$ *and* $[\mathit{During}(j, i)]$ *are* $\Diamond(\in_i \wedge \neg\in_j \wedge \Diamond(\in_i \wedge \in_j \wedge \Diamond(\in_i \wedge \neg\in_j)))$;
- $[\mathit{Starts}(i, j)]$ *and* $[\mathit{StartedBy}(j, i)]$ *are*
  $\Box(\in_i \Rightarrow \in_j) \wedge \neg\Diamond(\in_j \wedge \neg\in_i \wedge \Diamond\in_i) \wedge \Diamond(\in_j \wedge \neg\in_i)$;
- $[\mathit{Ends}(i, j)]$ *and* $[\mathit{EndedBy}(j, i)]$ *are*
  $\Box(\in_i \Rightarrow \in_j) \wedge \Diamond(\in_j \wedge \neg\in_i) \wedge \neg\Diamond(\in_j \wedge \in_i \wedge \Diamond(\in_j \wedge \neg\in_i))$;
- $[\mathit{Holds}(p, i)]$ *is* $\Box(\in_i \Rightarrow p)$; *and*
- $[\mathit{Occurs}(p, i)]$ *is* $[\neg\mathit{Holds}(\neg p, i)]$, *that is,* $\Diamond(\in_i \wedge p)$.

Example 2. Let us consider again the subformula

$$(\mathit{Meets}(N_{hb}, G_b) \wedge \mathit{Meets}(G_b, H_b) \wedge \mathit{During}(G_b, @(\mathit{tree})) \wedge \mathit{During}(G_b, H))$$

of the formula that characterizes the compatibilities of the monkey bananas problem (see Example 1), to illustrate how to encode an ALTL formula into an equivalent $\mathsf{LTL}_\Box$ one. Its encoding is:

$$\begin{array}{l}
\Diamond(\in_{N_{hb}} \wedge \Diamond\in_{G_b} \wedge \neg\Diamond(\in_{N_{hb}} \wedge \in_{G_b}) \wedge \neg\Diamond(\neg\in_{N_{hb}} \wedge \neg\in_{G_b} \wedge \Diamond\in_{G_b}))\wedge \\
\Diamond(\in_{G_b} \wedge \Diamond\in_{H_b} \wedge \neg\Diamond(\in_{G_b} \wedge \in_{H_b}) \wedge \neg\Diamond(\neg\in_{G_b} \wedge \neg\in_{H_b} \wedge \Diamond\in_{H_b}))\wedge \\
\Diamond(\in_{@(tree)} \wedge \neg\in_{G_b} \wedge \Diamond(\in_{@(tree)} \wedge \in_{G_b} \wedge \Diamond(\in_{@(tree)} \wedge \neg\in_{G_b})))\wedge \\
\Diamond(\in_H \wedge \neg\in_{G_b} \wedge \Diamond(\in_H \wedge \in_{G_b} \wedge \Diamond(\in_H \wedge \neg\in_{G_b}))) \wedge (\bigwedge_{i \in \mathcal{I}} \psi_i),
\end{array}$$

where $\mathcal{I} = \{N_{hb}, H_b, H, G_b, @(\mathit{tree})\}$ and ψ_i is $\Diamond\in_i \wedge \neg\Diamond(\in_i \wedge \Diamond(\neg\in_i \wedge \Diamond\in_i))$. As expected, the LTL encoding of the entire formula in Example 1 is very large. □

Appendix A discusses an implementation of this encoding based on term rewriting using the Maude [4] system.

Theorem 1. *Given an* $\mathsf{ALTL}(\mathcal{P}, \mathcal{I})$ *formula* φ *and a* $(\mathcal{P}, \mathcal{I})$*-interval model* $\mathcal{M}$*, then* $\mathcal{M} \models_{\mathsf{ALTL}} \varphi$ *iff* $[\mathcal{M}] \models_{\mathsf{LTL}} [\varphi]$.

Proof. Structural induction on φ. If φ has the form $\neg\varphi_1$ then $\mathcal{M} \models_{\mathsf{ALTL}} \varphi$ is equivalent to saying that it is *not* the case that $\mathcal{M} \models_{\mathsf{ALTL}} \varphi_1$, which, by the induction hypothesis and Definition 7, is equivalent to saying that $[\mathcal{M}] \models_{\mathsf{LTL}} [\varphi]$. The case where φ has the form $\varphi_1 \wedge \varphi_2$ is similar. What is left to show is that the property holds when φ is any of the interval relations. Let us discuss only one of them, for example $\mathit{Meets}(i, j)$. Suppose that $\mathcal{M} = (T, <, v, \sigma)$ and recall that $\sigma(i)$ is non-empty for any interval i. By Definition 4, $\mathcal{M} \models_{\mathsf{ALTL}} \mathit{Meets}(i, j)$

iff $\sigma(i) < \sigma(j)$ and there is no $t \in T$ such that $\sigma(i) < t < \sigma(j)$. By the way $[\mathcal{M}]$ is built and because ψ_i and ψ_j ensure the non-emptiness and the convexity of the trace fragments in which $\in_i$ and $\in_j$ hold, This is equivalent to saying that $\in_j$ holds *strictly* after $\in_i$, i.e., the $\Diamond(\in_i \wedge \Diamond \in_j \wedge \neg\Diamond(\in_i \wedge \in_j) \wedge ...)$ part of $[\mathit{Meets}(i,j)]$, and that there is no period of time following $\in_i$ that appears before $\in_j$ in which neither $\in_i$ nor $\in_j$ holds, i.e., the $\Diamond(...\neg\Diamond(\neg\in_i \wedge \neg\in_j \wedge \Diamond\in_j))$ part of $[\mathit{Meets}(i,j)]$. The result can be proved similarly for the other intervals. □

Our goal next is to reduce the satisfiability problem for ALTL to $\mathsf{LTL}_\Box$ satisfiability, known to be an NP-complete problem [15]. Theorem 1 gives us only half of the result, namely that if a formula φ is satisfiable in ALTL then the formula $[\varphi]$ is satisfiable in $\mathsf{LTL}_\Box$. To get the other half, one could define a slightly different translation of ALTL formulae, namely one that would also include the conjunction of the formulae in $\Psi_\mathcal{I}$. The problem with that is, however, that $\mathcal{I}$ can be infinite, meaning that the generated LTL formula would be infinite. Fortunately, only the intervals that explicitly appear in φ need to be taken into account, thus making our transformation finite:

Definition 8. *For an $\mathsf{ALTL}(\mathcal{P},\mathcal{I})$ formula φ, let $\mathcal{I}_\varphi$ be the finite set of intervals appearing in φ and let $\langle\varphi\rangle$ be the formula $[\varphi] \wedge \bigwedge \Psi_{\mathcal{I}_\varphi}$ in $\mathsf{LTL}(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}_\varphi\})$.*

Proposition 2. *Given a formula φ in $\mathsf{ALTL}(\mathcal{P},\mathcal{I})$, the following are equivalent:*

(1) φ is satisfiable in $\mathsf{ALTL}(\mathcal{P},\mathcal{I})$;
(2) $\langle\varphi\rangle$ is satisfiable in $\mathsf{LTL}(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}_\varphi\})$; and
(3) $\langle\varphi\rangle$ is satisfiable in $\mathsf{LTL}(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}\})$.

Proof. Since a model over more atomic propositions can be also regarded as a model over fewer propositions, it is immediate that *(3)* implies *(2)*. By Theorem 1, any model of φ in $\mathsf{ALTL}(\mathcal{P},\mathcal{I})$ yields a model of $[\varphi]$ in $\mathsf{LTL}(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}\})$ that satisfies $\Psi_\mathcal{I}$. Therefore, *(1)* implies *(3)*. To show that *(2)* implies *(1)*, by Proposition 1 it suffices to show that any model in $\mathsf{LTL}(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}_\varphi\})$ satisfying $\Psi_{\mathcal{I}_\varphi}$ can be extended, by just adding appropriate valuations for the additional atomic propositions to assure that the satisfaction of φ is not affected, to a model in $\mathsf{LTL}(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}\})$ satisfying $\Phi_\mathcal{I}$. This can be done many different ways. One straightforward model extension is to require that each proposition in $\{\in_i \mid i \in \mathcal{I} - \mathcal{I}_\varphi\}$ holds in precisely one (arbitrary) time point.

Corollary 1. *The satisfiability problem for ALTL is NP-complete.*

Proof. By Proposition 2, an ALTL formula φ is satisfiable iff $\langle\varphi\rangle$ is satisfiable as an LTL formula. Since $\langle\varphi\rangle$ can be generated linearly in the size of the φ and since LTL-satisfiability is NP-complete, ALTL-satisfiability is also NP-complete.

4 Monitoring ALTL

In this section we address the following two questions: (1) What is the monitoring complexity for ALTL? (2) Can we find an effective monitoring algorithm for ALTL? Regarding the first question we show that, despite the fact that the monitoring problem for LTL is EXPSPACE-complete [14], the monitoring problem for ALTL is NP-complete. Regarding the second, we first argue that a blind use of monitoring algorithms for LTL may be unfeasible in large applications and then propose an ALTL-specific monitoring algorithm which avoids the complexity of monitoring LTL-formulae. More precisely, we give a monitoring algorithm for ALTL whose most expensive task is to check the satisfiability of a boolean formula that is incrementally smaller (in the sense that some of its variables are irreversibly replaced by true or false) with each event received from the monitored system, and which initially has precisely the size of the original ALTL formula. Thus, the answer to (1) follows as a corollary to our solution to (2).

Let us first describe the "monitoring problem" for LTL. Given an LTL formula ξ of size n and a "running system" abstracted by its incrementally emitted events (or abstract states encoded by the atomic propositions that "hold" in them) t_1, t_2, ..., the problem is to report when a bad prefix is reached, that is, when a finite trace $t_1 t_2 .. t_m$ is encountered such that there is no infinite trace $t_1 t_2 .. t_m t_{m+1} ..$ that satisfies ξ. We here assume that storing the events is *not* an option. If in a particular application storing the events *is* feasible, then one can traverse them backwards each time a new event t_m is generated using a dynamic programming algorithm [14] and answer the problem polynomially with m and n; however, note that m can be large enough so that an algorithm linear in the continuously increasing execution trace at each emitted event can become easily more impractical than one just exponential in the formula but constant in the trace (e.g., when one generates an automata monitor from it, like in [2]).

One can reduce the satisfiability problem of any logic to a *synchronous monitoring* (i.e., violations are reported immediately) problem: given a formula φ, a monitor should report violation on the empty trace iff φ is not satisfiable. Thus, the fact that monitoring LTL is EXPSPACE-complete [14] comes at no surprise (LTL satisfiability is PSPACE [15]). Since ALTL is NP-complete (Corollary 1), any monitoring algorithm for ALTL is expected to be worst-case exponential in practice. However, as in many other similar situations, this fact does not necessarily mean that the problem of monitoring ALTL formulae is not practical. We next briefly discuss an immediate monitoring algorithm for ALTL based on its translation into LTL, and then give an algorithm specific to ALTL that avoids the complexity of monitoring LTL and which seems quite efficient in practice. The next section discusses an experiment where the ALTL formula is large enough that the LTL-based monitoring algorithms for ALTL cannot handle it.

The transformation in Section 3 suggests using a general purpose monitoring algorithm for LTL (e.g., the one in [2]), to monitor the LTL formula obtained linearly from the ALTL formula. We have experimented with this technique and have succeeded to generate, unfortunately huge, LTL monitors only for relatively small ALTL formulae. For example, for the ALTL formula in Example 2, which is a subformula of the ALTL formula in Example 1, the generated monitor had more than 60,000 edges, while the algorithm ran out of memory trying to generate an LTL monitor for the entire ALTL formula in Example 1; and that is just a toy example. The reason for our failure to generate monitors following this approach is the intermediate Buchi automata generator from LTL formulae; the LTL monitors in [2] are obtained by pruning the corresponding Buchi automata, which can be exponential in the size of the LTL formula. The interested reader is encouraged to check Appendix A for more details on this unsuccessful approach.

We next give a monitoring algorithm for ALTL *not* based on general monitoring algorithms for LTL. The idea is to regard the ALTL formula φ as a *boolean proposition* in which the interval relations are regarded as special variables. For each interval relation we generate a little state machine, which has two special states, true and false. These state machines are shown in Figure 2. We also add a top-level conjunct consisting of precisely one special variable for each interval that appears in φ; these latter variables correspond, intuitively, to the formulae ψ_i in Definition 5. The monitoring algorithm works as follows:

(1) generate all the state machines in Figure 2 (left-top state is initial);
(2) let ξ be the boolean proposition obtained from φ as above;
(3) run a *boolean* satisfiability checker on ξ and halt with "error" if ξ is not satisfiable;
(4) otherwise, for the next event t received from the monitored system, run all the state machines one step according to t (take that deterministic edge which is satisfied by t);
(5) modify the formula ξ by replacing each variable whose corresponding state machine is in a state true or false by the corresponding truth value;
(5) goto step (3).

Let us briefly discuss the state machines. The ones for ψ_i ensure that intervals are contiguous (convex); some intervals can be unbounded. The next seven state machines correspond to the relations on intervals. Let us discuss the one for $\mathit{Meets}(i,j)$. One starts with the initial state $\textcircled{\bar{i},\bar{j}}$ (neither in i nor in j), and there it stays as far as one does not enter any of the intervals. If while in this state the monitored program enters the interval j, that is, if $\in_j$ holds, then the relation $\mathit{Meets}(i,j)$ is obviously violated (interval i cannot be empty). Otherwise, if the interval i but not j is entered, then the machine moves to state $\textcircled{i,\bar{j}}$ where it waits until either i is left and j is entered in which case it returns true, or otherwise until i is left without entering j or i and j overlap, when it returns false. The machine for $\mathit{Holds}(p,i)$ checks that p holds during the interval i.

Example 3. Let us consider again the monkey/banana formula in Example 2,

$$(\mathit{Meets}(N_{hb}, G_b) \wedge \mathit{Meets}(G_b, H_b) \wedge \mathit{During}(G_b, @(\mathit{tree})) \wedge \mathit{During}(G_b, H)),$$

and consider an execution trace which starts with the abstract events $t_1 = \{\in_{N_b}\}$, $t_2 = \{\in_{N_b}, \in_{@(\mathit{tree})}\}$, $t_3 = \{\in_{G_b}, \in_{@(\mathit{tree})}, \in_H\}$, $t_4 = \{\in_{H_b}, \in_H\}$, ..., where an abstract event formed of a set of atomic propositions is an event in which all those, and only those propositions hold. This execution trace corresponds to the "flying monkey" scenario at the end of Example 1.

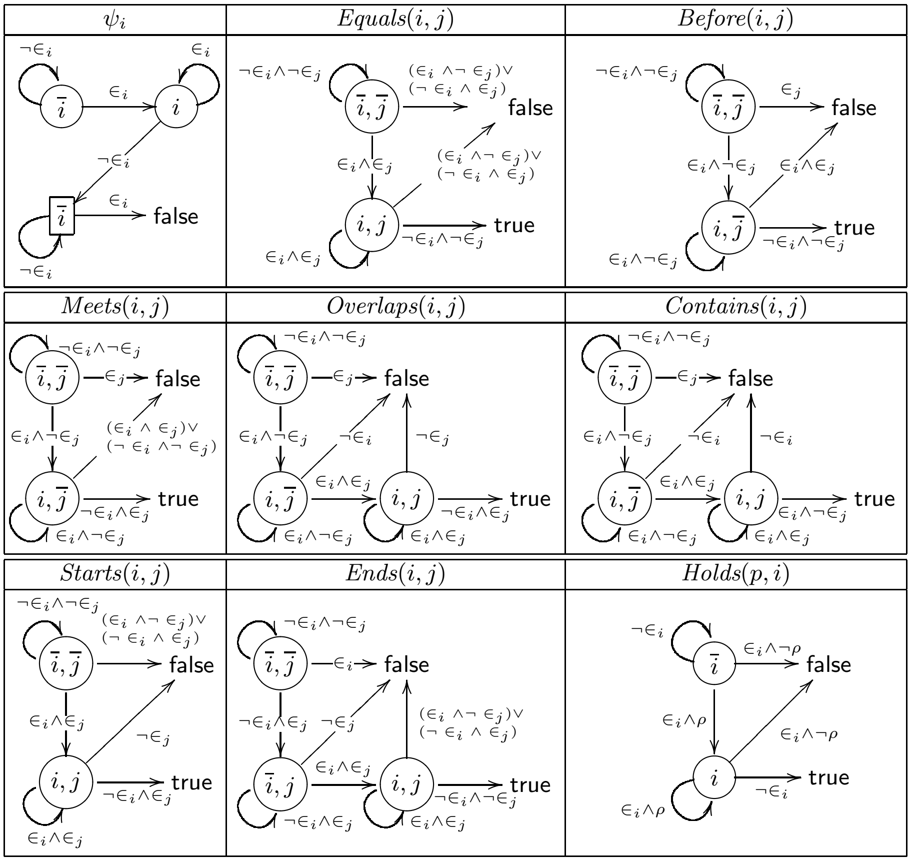


Fig. 2. State machines are run synchronously by the monitor with each event.

Let us simulate the execution of the ALTL monitoring algorithm above on this example. There are nine state machines like in Figure 2 necessary, four corresponding to each of the four interval relations and five corresponding to each interval appearing in the formula. The boolean formula ξ is just a conjunction of the corresponding nine variables. All one needs to do is to run the nine state machines on the execution trace, update the boolean proposition and then check for satisfiability after each event. After the first three events, the five ψ_i formulae will be in some intermediate (not false) states, and the four machines corresonding to the interval relations will be in the states true, $(G_b, \overline{H_b})$, $(@(tree), G_b)$, and (G_b, H), respectively, so the formula is still satisfiable. However, when the event t_4 is processed, the machine corresponding to $During(G_b, @(tree))$, or to $Contains(@(tree), G_b)$, transits to false, invalidating the boolean proposition. □

Example 4. Consider now the ALTL formula $\neg \mathit{Before}(i,j)$ and a two-event trace $\{\in_i\}\{\}$. The monitoring algorithm above sets the machine corresponding to $\mathit{Before}(i,j)$ to state $\enclose{circle}{i,\bar{j}}$ after processing $\{\in_i\}$ and then to state true after processing $\{\}$, causing the monitor to report "error" before any event containing $\in_j$ is seen. Note that $\{\in_i\}\{\}$ is indeed a bad prefix for $\neg \mathit{Before}(i,j)$ ($\in_j$ must hold eventually in any interval model of ALTL). Therefore, our monitoring algorithm for ALTL detects bad prefixes as soon as they appear. □

Theorem 2. *The monitoring algorithm for ALTL above is correct.*

Proof. First, note that the state machines corresponding to ψ_i and "running" at the top of the boolean proposition ξ will intercept any violation of the convexity of intervals. If any of the convexities of intervals is violated, that is, if an interval starts, then it is interrupted and then started again, then the monitoring algorithm above returns "error", because the observed trace cannot even be continued into an interval model; one can easily modify the algorithm to return a different type of error in such situations. Hence, from now on in the proof we assume the well-formedness of intervals. Consider some finite trace $\tau = t_1 t_2 ... t_m$ that is well-formed wrt intervals, i.e., it can be the prefix of some interval model of ALTL. Let us first prove that for any interval relation, its corresponding state machine is in state false after processing τ iff τ is a bad prefix of that interval relation. We only show it for one relation, say $\mathit{Before}(i,j)$; the others are similar. Note that τ is a bad prefix of $\mathit{Before}(i,j)$ iff τ contains (some event satisfying) $\in_j$ before or at the same time with $\in_i$. Since the state machine of $\mathit{Before}(i,j)$ reaches the state false iff $\in_j$ is seen before $\in_j$ or if $\in_j$ and $\in_i$ are seen together as part of an event, and since the machines corresponding to ψ_i ensure the contiguity of intervals, we can conclude that τ is a bad prefix of $\mathit{Before}(i,j)$ iff the corresponding machine of $\mathit{Before}(i,j)$ is in state false after processing τ.

Let us next prove that for any interval relation, the corresponding machine is in state true after processing τ iff τ is a good prefix of that relation, in the sense that for any infinite trace τ such that $\tau\pi$ is an interval model of ALTL, it is the case that $\tau\pi$ satisfies that relation. As above, let us just prove it for $\mathit{Before}(i,j)$. Note that the machine of $\mathit{Before}(i,j)$ can be in state true after processing τ iff τ contains no event satisfying $\in_j$ and contains some event satisfying $\in_i$ followed by one which does not satisfy $\in_i$. This is equivalent to saying that any interval model of the form $\tau\pi$ (recall that intervals have non-empty interpretations in interval models) satisfies $\mathit{Before}(i,j)$.

Let us now consider any ALTL formula φ and a finite trace τ as above. If φ has the form $\varphi_1 \wedge \varphi_2$ then τ is a bad (or good) prefix of φ iff it is a bad (or good prefix) of φ_1 or (and) φ_2. If φ has the form $\neg\varphi_1$ then τ is a bad (or good) prefix of φ iff it is a good (or a bad) prefix of φ_1. Therefore, in order to test whether τ is a bad prefix of φ one only needs to know whether it is a bad prefix of φ's interval relations, that is, if their corresponding state machines are in their corresponding false or true states after processing τ. The satisfiability checking of ξ after each event ensures that violations are reported as early as possible. □

Corollary 2. *The monitoring problem for ALTL is NP-complete.*

If one is not interested in reporting ALTL property violations as early as possible, then one can run the satisfiability checker less frequently, say once every 100 evens, or even just once at the end of the monitoring session, and thus significantly reduce the runtime overhead (which would amount to just running the state machines in parallel at each received event). If minimal runtime overhead is highly desirable, since the formula ξ to check for satisfiability changes incrementally by irreversibly transforming some of its variables into true or false, to achieve a minimal runtime overhead one can use an incremental SAT solver.

5 Experiment

Implementation. We have implemented a prototype monitor generation tool, called ALTL2Monitor. It implements the monitoring algorithm presented in the previous section using the SAT solver zChaff [11] for satisfiability checking.
Case Study. Our case study is a simplified version of an exploration rover (Gromit, at Nasa Ames). The mission of the robot is to visit a number of waypoints, into an initially unknown rough environment, while monitoring interesting targets on its path. The robot continuously takes pictures of the terrain in front of it, performs a stereo correlation to extract cloud of 3D points, merges these points in its model of environment and starts this process again. In parallel, it continuously considers its currents position, the next waypoint to visit, the obstacles in the model of the environment built and produces a trajectory. These two interdependent cyclic processes are synchronized. Last, a third process interrupts whenever an interesting rock has been detected. The functional layer of Gromit is implemented using functional modules (for more details see [10]). For each of them we shall consider the "visible" state variables of interest :
- Rflex is the module interfaced with the low-level speed controller. It has a state variable for the position of the robot, with each token representing a specific robot position, and another one for the speed passed to the wheels controller.
- Camera shots a pair of stereo calibrated images and saves them. It has one state variable representing the camera status (taking picture, or idle).
- SCorrel produces and stores a stereo correlated image. It has a state variable representing the Scorrel process (performing stereo correlation, or idle).
- Lane builds a model of the environment by aggregating clouds of 3D points produced by Scorrel. It services two requests: read in an internal buffer and fuse the read. Lane has one state variable for the model building process.
- P3D is a rover navigation software. It produces an arc trajectory which is translated in a speed reference, to try to reach a waypoint. P3D has a variable for its state (idle or computing the speed) and one for the waypoints to visit.
- Science. This module monitors a particular condition of interest to scientist (such as detecting a rock with particular features). When such a condition arises while the robot is moving toward a waypoint, it stops and takes a picture of the rock. It has one state variable for its state (monitoring interesting rock or idle).

Figure 3 shows some temporal relations representing a simplified version of the actual Gromit Rover.

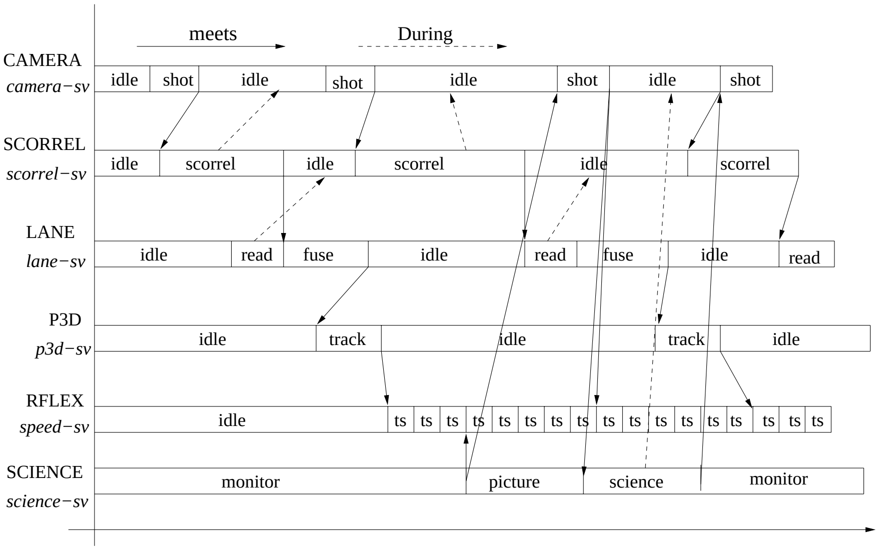


Fig. 3. Partial Gromit Model: Attributes and compatibilities

Results. Due to intellectual property restrictions, we did not have access to the execution platform of the Gromit Rover. However, the CNRS Laboratory LAAS (at Toulouse, France) provided[1] us with a file formalizing some of the compatibilities as an ATL formula of more than 100 interval relations, listed in Appendix B, as well as with a set of one hundred traces generated by the Gromit Rover execution platform. We applied our prototype ALTL2Monitor off-line to check these traces; the checking took negligible time. However, the satisfiability checker was applied only once at the end of the monitoring session of each trace, because we expected the traces to be correct, which was indeed the case.

6 Conclusion

We presented Allen linear temporal logic (ALTL), an automated translation of ALTL into LTL, a monitor synthesis algorithm for ALTL, as well as a real-life experiment. While LTL can be a suitable logic for AI and planning, we also believe that ALTL can be a suitable logic for certain program verification efforts. Its simplicity and neutrality cannot be ignored, while at the same time tends to be algorithmically more efficient than LTL; for example, the corresponding LTL-formula to the ALTL formula in the experiment above would have hundreds of nested temporal operators, with little or no hope to generate a Buchi automaton for it. We plan to apply our ALTL monitoring prototype to the autonomous embedded system iRobot ATRV of the LAAS Laboratory.

[1] We warmly thank Felix Ingrand for help.

References

1. J. Allen. Towards a general theory of actions and time. *Artificial Intelligence*, 23(2):123–154, 1984.
2. M. Amorim and G. Roşu. Efficient monitoring of omega-languages. In *CAV'05*, volume 3576 of *LNCS*, pages 364–378. Springer, July 2005.
3. D. Calvanese, G. De Giacomo, and M. Y. Vardi. Reasoning about actions and planning in LTL action theories. In *Knowledge Representation and Reasoning (KR'02)*, pages 593–602, 2002.
4. M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer, and C.L. Talcott. Maude, a rewrite logic system. http://maude.cs.uiuc.edu/.
5. M. Ghallab and A.M. Alaoui. Managing efficiently temporal relations through indexed spanning trees. In *International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 1297–1303, 1989.
6. J.A. Goguen and G. Roşu. Institution morphisms. *J. Formal Aspects of Computing*, 13(3-5):274–307, 2002.
7. A. Jonsson and J. Frank. A framework for dynamic constraint reasoning using procedural constraints. In *Workshop on Constraints in Control, part of the 5th International Conference On Principles and Practice of Constraint Programming,(CP'99)*, 1999.
8. A. Jonsson, P. Morris, N. Muscettola, K. Rajan, and B. Simth. Planning in interplanetary space: theory and practice. In *Fifth International Conference on Artificial Intelligence Planning Systems (AIPS'00)*, Breckenridge, Colorado, 2000.
9. A.A. Krokhin, P. Jeavons, and P. Jonsson. Reasoning about temporal relations: The tractable subalgebras of allen's interval algebra. *J. ACM*, 50(5):591–640, 2003.
10. S. Lacroix, A. Mallet, D. Bonnafous, G. Bauzil, S. Fleury, M. Herrb, and R. Chatila. Autonomous rover navigation on unknown terrains, functions and integration. *International Journal of Robotics Research*, 21(10-11):917–942, 2002.
11. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference (DAC'01)*, pages 530–535. IEEE, June 2001.
12. Dennis Oddoux and Paul Gastin. LTL2BA, a Buchi automata generator for LTL. `http://www.liafa.jussieu.fr/~oddoux/ltl2ba/`.
13. A. Pnueli. The temporal logic of programs. In *Proceedings of Foundations of Computer Science (FOCS'77)*, New York, 1977. IEEE.
14. G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *J. of Automated Software Engineering*, 12(2):151–197, 2005.
15. A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.
16. M. Vilain, H. Kautz, and P. van Beek. Constraint propagation algorithms for temporal reasoning: a revised report, 1989. In *Readings in Qualitative Reasoning about Phisical Systems*, Morgan Kaufmann, Los Altos, CA.

A A Rewriting-based Translator from ALTL into LTL

In this appendix we encode the translation of ALTL formulae into LTL formulae discussed in Section 3 using the Maude [4] rewrite engine. We start with a module declaring a sort `Proposition` for atomic propositions. Technically speaking, we could have just used the sort `Formula` (see next module) instead of `Proposition`, but the definition would be harder to read:

```
fmod PROPOSITION is
  sort Proposition .
endfm
```

The next module adds the sort `Formula` for formulae together the syntax for the usual boolean operators:

```
fmod BASIC-FORMULA is
  including PROPOSITION .
  sort Formula .
  subsort Proposition < Formula .
  ops true false : -> Formula .
  op _/\_ : Formula Formula -> Formula [assoc prec 20] .
  op _\/_ : Formula Formula -> Formula [assoc prec 30] .
  op _->_ : Formula Formula -> Formula [prec 40] .
  op _<->_ : Formula Formula -> Formula [prec 40] .
  op !_ : Formula -> Formula [prec 10] .
endfm
```

We next add syntax for the LTL$_\Box$ operators:

```
fmod LTL-BOX-SYNTAX is
  including BASIC-FORMULA .
  ops <>_ ‘[‘]_ : Formula -> Formula [prec 7] .
endfm
```

We next add the sort `Interval` for intervals:

```
fmod INTERVAL is
  sort Interval .
endfm
```

Now we are ready to define the translator from ALTL into LTL. Recall that Maude interprets all the equations into parametric rewrite rules, that is, they are only applied from left to right when one launches its rewrite engine. We first give reduction rules for all the derived interval relations:

```
fmod ALLEN-SYNTAX is
  including BASIC-FORMULA .
  including INTERVAL .
  ops Equals Before After Overlaps Overlapped-by
      Meets Met-by Contains During
      Starts Started-by Ends Ended-by : Interval Interval -> Formula .
  ops Holds Occurs : Proposition Interval -> Formula .
  var P : Proposition .  vars I J : Interval .
  eq After(I,J) = Before(J,I) .
  eq Met-by(I,J) = Meets(J,I) .
  eq Overlapped-by(I,J) = Overlaps(J,I) .
  eq During(I,J) = Contains(J,I) .
  eq Started-by(I,J) = Starts(J,I) .
  eq Ended-by(I,J) = Ends(J,I) .
endfm
```

To define the actual translation, we need all the intervals in a formula in order to generate the LTL formulae ψ_i (see Section 3). To achieve that, we define sets of intervals first:

```
fmod INTERVAL-SET is
  including INTERVAL .
  sort IntervalSet .
  subsort Interval < IntervalSet .
  op empty : -> IntervalSet .
  op _,_ : IntervalSet IntervalSet -> IntervalSet [assoc comm id: empty] .
  var I : Interval .
  eq I,I = I .
endfm
```

Then we define a (boring) operation, `intervalsOf`, which accumulates all the intervals appearing in a given ALTL formula in a corresponding set:

```
fmod INTERVALS-OF is
  including ALLEN-SYNTAX .
  including INTERVAL-SET .
  op intervalsOf : Formula -> IntervalSet .
  vars I J : Interval .  vars F F’ : Formula .  var P : Proposition .
  eq intervalsOf(Equals(I,J)) = I,J .
  eq intervalsOf(Before(I,J)) = I,J .
  eq intervalsOf(Overlaps(I,J)) = I,J .
  eq intervalsOf(Meets(I,J)) = I,J .
  eq intervalsOf(Contains(I,J)) = I,J .
  eq intervalsOf(Starts(I,J)) = I,J .
  eq intervalsOf(Ends(I,J)) = I,J .
  eq intervalsOf(Holds(P,I)) = I .
  eq intervalsOf(Occurs(P,I)) = I .
  eq intervalsOf(true) = empty .
  eq intervalsOf(false) = empty .
  eq intervalsOf(F /\ F’) = intervalsOf(F), intervalsOf(F’) .
  eq intervalsOf(F \/ F’) = intervalsOf(F), intervalsOf(F’) .
  eq intervalsOf(F -> F’) = intervalsOf(F), intervalsOf(F’) .
  eq intervalsOf(! F) = intervalsOf(F) .
endfm
```

The following is now straightforward; it blindly implements the translation rules in Section 3:

```
fmod ALLEN2LTL is
  including LTL-BOX-SYNTAX .
  including INTERVALS-OF .
  op in : Interval -> Proposition .
  op psi : IntervalSet -> Formula .
  vars I J : Interval .  var Is : IntervalSet .
  eq psi(I) = <> in(I) /\ ! <>(in(I) /\ <>(! in(I) /\ <> in(I))) .
  eq psi(I,J,Is) = psi(I) /\ psi(J,Is) .

  ops `[_`] `{_`} : Formula -> Formula .
  var P : Proposition .  vars F F' : Formula .
  eq [Equals(I,J)] = [](in(I) <-> in(J)) .
  eq [Before(I,J)] = <>(in(I) /\ <>(! in(I) /\ ! in(J) /\ <> in(J))) .
  eq [Meets(I,J)]
   = <>(in(I) /\ <> in(J) /\ ! <>(in(I) /\ in(J)) /\ ! <>(! in(I) /\ ! in(J) /\ <> in(J))) .
  eq [Overlaps(I,J)]
   = <>(in(I) /\ ! in(J) /\ <>(in(I) /\ in(J) /\ <>(! in(I) /\ in(J)))) .
  eq [Contains(I,J)]
   = <>(in(I) /\ ! in(J) /\ <>(in(I) /\ in(J) /\ <>(in(I) /\ ! in(J)))) .
  eq [Starts(I,J)]
   = [](in(I) -> in(J)) /\ ! <>(in(J) /\ ! in(I) /\ <> in(I)) /\ <>(in(J) /\ ! in(I)) .
  eq [Ends(I,J)]
   = [](in(I) -> in(J)) /\ <>(in(J) /\ ! in(I)) /\ ! <>(in(J) /\ in(I) /\ <>(in(J) /\ ! in(I))) .
  eq [Holds(P,I)] = [](in(I) -> P) .
  eq [Occurs(P,I)] = <>(in(I) /\ P) .
  eq [true] = true .
  eq [false] = false .
  eq [F /\ F'] = [F] /\ [F'] .
  eq [F \/ F'] = [F] \/ [F'] .
  eq [! F] = ! [F] .

  eq {F} = [F] /\ psi(intervalsOf(F)) .
endfm
```

The encoding is done. We can now try various examples, such as the one in Example 1:

```
fmod TEST is
  including ALLEN2LTL .
  ops a b c d x y z : -> Proposition .
  ops i j k : -> Interval .

  op hungry : -> Proposition .
  ops Nhb Gb Hb @tree H @x Gx2tree L C CD : -> Interval .
  ops formula1 formula2 formula3 formula4 : -> Formula .

  eq formula1 = Meets(Nhb,Gb) /\ Meets(Gb,Hb) /\ During(Gb,@tree) /\ During(Gb,H) .
  eq formula2 = Meets(@x,Gx2tree) /\ Meets(Gx2tree,@tree) /\ During(Gx2tree,L) .
  eq formula3 = Meets(L,C) /\ Meets(C,H) /\ Meets(H,CD) /\ Meets(CD,L) /\ During(C,@tree) .
  eq formula4 = Occurs(hungry,Nhb) /\ Holds(hungry,Gb) /\ Holds(hungry,Hb) .
endfm

reduce {formula1 /\ formula2 /\ formula3 /\ formula4} .
```

Maude produces the following result:

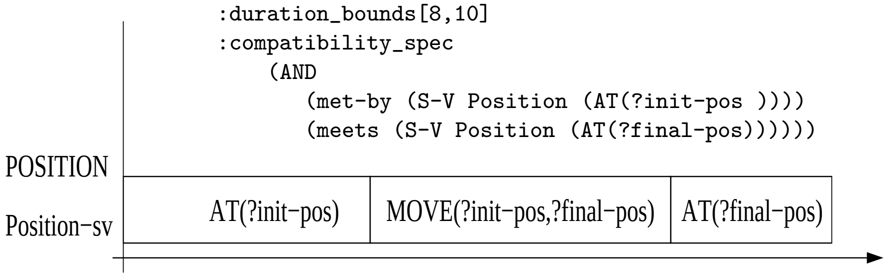
```
bash$ maude allen2ltl.maude
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.1.1 built: Jun 15 2004 12:55:31
             Copyright 1997-2004 SRI International
                   Mon Jan 23 16:18:08 2006
==========================================
reduce in TEST : {formula1 /\ formula2 /\ formula3 /\ formula4} .
rewrites: 103 in 0ms cpu (0ms real) (~ rewrites/second)
result Formula: <> (in(Nhb) /\ <> in(Gb) /\ ! <> (in(Nhb) /\ in(Gb))
/\ ! <> (! in(Nhb) /\ ! in(Gb) /\ <> in(Gb))) /\ <> (in(Gb) /\
<> in(Hb) /\ ! <> (in(Gb) /\ in(Hb)) /\ ! <> (! in(Gb) /\ ! in(Hb) /\
<> in(Hb))) /\ <> (in(@tree) /\ ! in(Gb) /\ <> (in(@tree) /\ in(Gb) /\
<> (in(@tree) /\ ! in(Gb)))) /\ <> (in(H) /\ ! in(Gb) /\ <> (in(H) /\
in(Gb) /\ <> (in(H) /\ ! in(Gb)))) /\ <> (in(@x) /\ <> in(Gx2tree) /\
! <> (in(@x) /\ in(Gx2tree)) /\ ! <> (! in(@x) /\ ! in(Gx2tree) /\
<> in(Gx2tree))) /\ <> (in(Gx2tree) /\ <> in(@tree) /\ ! <> (in(Gx2tree)
/\ in(@tree)) /\ ! <> (! in(Gx2tree) /\ ! in(@tree) /\ <> in(@tree)))
/\ <> (in(L) /\ ! in(Gx2tree) /\ <> (in(L) /\ in(Gx2tree) /\ <> (in(L)
/\ ! in(Gx2tree)))) /\ <> (in(L) /\ <> in(C) /\ ! <> (in(L) /\ in(C))
/\ ! <> (! in(L) /\ ! in(C) /\ <> in(C))) /\ <> (in(C) /\ <> in(H) /\
! <> (in(C) /\ in(H)) /\ ! <> (! in(C) /\ ! in(H) /\ <> in(H))) /\
<> (in(H) /\ <> in(CD) /\ ! <> (in(H) /\ in(CD)) /\ ! <> (! in(H) /\
! in(CD) /\ <> in(CD))) /\ <> (in(CD) /\ <> in(L) /\ ! <> (in(CD) /\
in(L)) /\ ! <> (! in(CD) /\ ! in(L) /\ <> in(L))) /\ <> (in(@tree) /\
! in(C) /\ <> (in(@tree) /\ in(C) /\ <> (in(@tree) /\ ! in(C)))) /\
<> (in(Nhb) /\ hungry) /\ [](in(Gb) -> hungry) /\ [](in(Hb) -> hungry)
/\ <> in(Nhb) /\ ! <> (in(Nhb) /\ <> (! in(Nhb) /\ <> in(Nhb))) /\
<> in(Gb) /\ ! <> (in(Gb) /\ <> (! in(Gb) /\ <> in(Gb))) /\ <> in(Hb)
/\ ! <> (in(Hb) /\ <> (! in(Hb) /\ <> in(Hb))) /\ <> in(@tree) /\
! <> (in(@tree) /\ <> (! in(@tree) /\ <> in(@tree))) /\ <> in(H) /\
! <> (in(H) /\ <> (! in(H) /\ <> in(H))) /\ <> in(@x) /\
! <> (in(@x) /\ <> (! in(@x) /\ <> in(@x))) /\ <> in(Gx2tree) /\
! <> (in(Gx2tree) /\ <> (! in(Gx2tree) /\ <> in(Gx2tree))) /\ <> in(L)
/\ ! <> (in(L) /\ <> (! in(L) /\ <> in(L))) /\ <> in(C) /\ ! <> (in(C)
/\ <> (! in(C) /\ <> in(C))) /\ <> in(CD) /\ ! <> (in(CD) /\
<> (! in(CD) /\ <> in(CD)))
Maude>
```

We have unsuccessfully tried to generate a monitor from the LTL formula above using the monitor generator in [2], which is based on a Buchi automata generator for LTL formulae. The monitor generator in [2] uses the LTL2BA procedure by Oddoux and Gastin in [12]. Unfortunately, the Buchi automaton for the formula above seems to be prohibitively large. We iteratively reduced the size of the LTL formula until LTL2BA succeeded to generate a monitor. More precisely, when the LTL formula is the one corresponding to the ALTL subformula `formula1`, that is, about 25% the size of the original formula, LTL2BA was able to generate a Buchi automaton of several thousand states and more than 60,000 edges! This little experiment suggests that using blindly LTL monitor generators to obtain monitors for ALTL may not be feasible in practice, especially because relatively small ALTL formule can result in large corresponding LTL formulae. Thus we decided to investigate possibilities to generate specialized monitors for ALTL and eventually developed the algorithm in Section 4.

B The Compatibilities of the Case Study

The modeling language used for the case study is the Domain Description Language (DDL) supported by the EUROPA planning technology [7], a direct descendant of the Planner/Scheduler that was part of NASA's Remote Agent [8]. DDL uses a LISP-based syntax. A DDL model is equivalent to defining a set of parallel timelines (one per state variable), sets of procedure types that can appear on each timeline over which a procedure can extend, temporal constraints between procedure intervals (the compatibilities), duration constraints and parametric constraints. An example of compatibilities definition in DDL extracted from our case study, as well as its graphical interpretation and corresponding ALTL formula, is shown in Figure 4. The rest of this appendix contains the entire formal DDL specification of our case study, as it was provided by the CNRS Laboratory LAAS from Toulouse, France.

```
(Define-compatibility
     ((S-V Position)
          (MOVE(?init-pos ?final-pos)))
       :duration_bounds[8,10]
       :compatibility_spec
           (AND
              (met-by (S-V Position (AT(?init-pos ))))
              (meets (S-V Position (AT(?final-pos))))))
```



$$MetBy(MOVE(\mathit{?init\text{-}pos}, \mathit{?final\text{-}pos}), AT(\mathit{?init\text{-}pos})) \wedge$$
$$Meets(MOVE(\mathit{?init\text{-}pos}, \mathit{?final\text{-}pos}), AT(\mathit{?final\text{-}pos}))$$

Fig. 4. An example of definition in DDL, graphical representation and ALTL-formula.

```
;;; -*- Mode: Lisp -*-
;;; File:
;;; Date:
;;;  CVS:

(Define_Constant *max_goal_idle_time* 40)
(Define_Constant *goal_cicle_step* 2)

;;; This model describes the necessary functionality for
;;; implementing IDEA on Gromit

;;; Objects
(Define_Object_Class Gromit_Class
  :state_variables
                   (
    (Controllable Start_Manip_SV)
    (Controllable Camera_SV)
    (Controllable CameraObs_SV)

    (Controllable SCorrel_SV)

    (Controllable STEO_SV)
    (Controllable Localize_Goal_SV)

    (Controllable Lane_SV)
    (Controllable Map_Goal_SV)

                    (Controllable Away_SV)
            (Controllable Platine_SV)
    (Controllable Rflex_speed_cntl_SV)

                    (Controllable Photo_obj_goal_SV)

    (Controllable Global_Goal_SV)
    (Controllable Load_Goal_SV)
))


; Start_Manip_camera_req

 (Define_Member_Values ((Gromit_Class Start_Manip_SV))
   (Start_Manip_idle
    Start_Manip_monitor
 )
)

(Define_Member_Values ((Gromit_Class Camera_SV))
  (Camera_idle
   Camera_shot
)
)

(Define_Member_Values ((Gromit_Class SCorrel_SV))
  (SCorrel_idle
   SCorrel_scorrel
)
)

(Define_Member_Values ((Gromit_Class STEO_SV))
  (STEO_idle
   STEO_read
   STEO_pos
)
)

(Define_Member_Values ((Gromit_Class Lane_SV))
  (Lane_idle
   Lane_read
   Lane_fuse
)
)

(Define_Member_Values ((Gromit_Class Localize_Goal_SV))
  (Localize_Goal_idle
   Localize_Goal_localize
   )
)

(Define_Member_Values ((Gromit_Class Map_Goal_SV))
  (Map_Goal_idle
   Map_Goal_map
   )
)

(Define_Member_Values ((Gromit_Class Global_Goal_SV))
  (Global_Goal_idle
   Global_Goal_exec
   Global_Goal_start
   Global_Goal_end
   )
```

```
)

(Define_Member_Values ((Gromit_Class Load_Goal_SV))
  (Load_Goal_idle
   Load_Goal_exec
   )
)



;;; Camera SVs


(Define_Member_Values ((Gromit_Class Away_SV))
   (
    Away_buff
    Away_idle
    Away_pos
 )
)


(Define_Member_Values ((Gromit_Class Platine_SV))
   (Platine_idle
    Platine_oriented
    Platine_pos
    Platine_back
 )
)

(Define_Member_Values ((Gromit_Class CameraObs_SV))
  (
   CameraObs_idle
   CameraObs_shot
   CameraObs_shotted
  )
)

(Define_Member_Values ((Gromit_Class Rflex_speed_cntl_SV))
   (Rflex_speed_cntl_idle
    Rflex_cntl
 )
)

(Define_Member_Values ((Gromit_Class Photo_obj_goal_SV))
   (Photo_obj_goal_start
    Photo_obj_goal_task
    Photo_obj_goal_idle
    Photo_obj_goal_end
 )
)

;;; Constants

(Define_Constant *max-dist-obstacle* 2)

(Define_Constant *latency* 1)

;;; Label sets

(Define_Label_Set ReturnStatus (OK PB))

(Define_Label_Set Return_Status (OK Failed))

(Define_Label_Set Task (Localize Observe))

(Define_Label_Set RflexCommand (ON OFF))

(Define_Label_Set GlobalExecStatus (Single))
;;; First Medium Last


;;; Define Predicate

(Define_Predicate Start_Manip_idle
(
 (Boolean xidle)
)
)

(Define_Predicate Start_Manip_monitor
   (
    (ReturnStatus status)
    (Boolean statusFlag)
   )
)

(Define_Predicate Camera_idle
  (
   (Integer shot_index)
   (Integer next_shot_index)
  )
)

(Define_Predicate Camera_shot
  (
   (Integer shot_index)
   (ReturnStatus status)
   (Boolean statusFlag)
  )
)

(Define_Predicate SCorrel_idle
  (
   (Integer shot_index)
  )
)

(Define_Predicate SCorrel_scorrel
  (
   (Integer shot_index)
   (ReturnStatus status)
   (Boolean statusFlag)
   )
  )

(Define_Predicate STEO_idle
  )


(Define_Predicate STEO_read
  (
   (Integer shot_index)
   (ReturnStatus status)
   (Boolean statusFlag)
   )
  )

(Define_Predicate STEO_pos
  (
   (Integer shot_index)
   (ReturnStatus status)
   (Boolean statusFlag)
   )
  )

(Define_Predicate Lane_idle
  )

(Define_Predicate Lane_read
  (
   (Integer shot_index)
   (ReturnStatus status)
   (Boolean statusFlag)
   )
  )

(Define_Predicate Lane_fuse
  (
   (Integer shot_index)
   (ReturnStatus status)
   (Boolean statusFlag)
   )
  )

(Define_Predicate Localize_Goal_idle
  )

(Define_Predicate Localize_Goal_localize
  (
   (Integer localize_index)
   )
  )

(Define_Predicate Map_Goal_idle
  )

(Define_Predicate Map_Goal_map
  (
   (Integer map_index)
   )
  )



(Define_Predicate Global_Goal_start
  )
```

```
(Define_Predicate Global_Goal_exec
  (
   (Integer gen)
   (Integer ngen)
   (GlobalExecStatus first)
   )
  )

(Define_Predicate Global_Goal_end
  (
   (Integer gen)
   )
)


;; (Define_Predicate  Global_Goal_idle2
;;   (
;;    (Integer prevgen)
;;    (Integer gen)
;;    )
;;)

(Define_Predicate  Global_Goal_idle
  (
   (Integer gen)
   )
  )

(Define_Predicate  Load_Goal_idle
  )

(Define_Predicate  Load_Goal_exec
  (
   (Integer gen)
   (Integer ngen)
   (ReturnStatus status)
   (Boolean statusFlag)
   )
)

; Free future
;(Define_Predicate Free_future_idle
;)

;(Define_Predicate Free_future_stretch
;)

;(Define_Predicate Free_future_now
; (
;   (Integer gen)
;   (Integer ngen)
;   (ReturnStatus status)
;   (Boolean statusFlag)
; )
;)



;; Camera predicates

;;; Away Perdicates

(Define_Predicate Away_buff
 (
   (Real xbuf)
 )
 )

(Define_Predicate Away_idle
 )

(Define_Predicate Away_pos
(
  (Real xpos)
  (Boolean xFlag)
  (ReturnStatus status)
  (Boolean statusFlag)
)
)


;;; Platine Predicates
(Define_Predicate Platine_pos
 (
   (Real ypos)
   (ReturnStatus status)
   (Boolean statusFlag)
 )
 )
```

```
(Define_Predicate Platine_idle
 )

 (Define_Predicate Platine_oriented
 )

(Define_Predicate Platine_back
 (
   (ReturnStatus status)
   (Boolean statusFlag)
 )
)

;; Rflex_speed
(Define_Predicate Rflex_speed_cntl_idle
  ((RflexCommand rcntl)))


(Define_Predicate Rflex_cntl
 (
   (RflexCommand ycntl)
   (ReturnStatus status)
   (Boolean statusFlag)
 )
)

;CameraObs

(Define_Predicate CameraObs_idle
)

(Define_Predicate CameraObs_shot
 (
   (ReturnStatus status)
   (Boolean statusFlag)
 )
)

(Define_Predicate CameraObs_shotted
)

; Goal predicates
(Define_Predicate Photo_obj_goal_task
)

(Define_Predicate Photo_obj_goal_idle
)

(Define_Predicate Photo_obj_goal_end
)

(Define_Predicate Photo_obj_goal_start
)

;;; Main constraints to enforce on the overall system


;;; post both Localize and Map goals.

(Define_Compatibility  (SINGLE ((Gromit_Class Start_Manip_SV))
       ((Start_Manip_idle(?xidle))))
  :duration_bounds [*latency* _plus_infinity_]
)

(Define_Compatibility  (SINGLE ((Gromit_Class Start_Manip_SV))
       ((Start_Manip_idle(?xidle))))
  :compatibility_spec
  (?xidle
   OR
   (True
    AND
     (meets
       (SINGLE ((Gromit_Class Start_Manip_SV))
            ((Start_Manip_monitor (?_any_value_)))))))
)

(Define_Compatibility  (SINGLE ((Gromit_Class Start_Manip_SV))
       ((Start_Manip_monitor (OK True))))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class Start_Manip_SV))
            ((Start_Manip_idle(?_any_value_))))))
)

(Define_Compatibility  (SINGLE ((Gromit_Class Start_Manip_SV))
       ((Start_Manip_monitor (OK True))))
  :compatibility_spec
  (AND
   (before [5 400]
```

```
    (SINGLE ((Gromit_Class Global_Goal_SV))
            ((Global_Goal_end (2)))))
   (meets
    (SINGLE ((Gromit_Class Away_SV))
            ((Away_pos (?_any_value_)))))
  )
)



;;; Localize
(Define_Compatibility  (SINGLE ((Gromit_Class Localize_Goal_SV))
       ((Localize_Goal_localize (?generation))))
  :duration_bounds [*latency* _plus_infinity_]
  :compatibility_spec
  (AND
   (met_by
    (SINGLE ((Gromit_Class Localize_Goal_SV))
    (Localize_Goal_idle)))
   (met_by
    (SINGLE ((Gromit_Class STEO_SV))
    ((STEO_pos (?generation OK True)))))
   (meets
    (SINGLE ((Gromit_Class Localize_Goal_SV))
    (Localize_Goal_idle)))
   ))

(Define_Compatibility  (SINGLE ((Gromit_Class Localize_Goal_SV))
       (Localize_Goal_idle))
  :duration_bounds [*latency* _plus_infinity_]
  :compatibility_spec
  (AND
   (met_by
    (SINGLE ((Gromit_Class Localize_Goal_SV))
    ((Localize_Goal_localize (?_any_value_)))))
   )
)

(Define_Compatibility  (SINGLE ((Gromit_Class STEO_SV))
       ((STEO_pos (?generation ?status ?statusFlag))))
  :duration_bounds [*latency* 6]
  :compatibility_spec
  (AND
   (met_by
   (SINGLE ((Gromit_Class STEO_SV))
   ((STEO_read (?generation OK True)))))
   ))

(Define_Compatibility  (SINGLE ((Gromit_Class STEO_SV))
       ((STEO_pos (?generation OK True))))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class STEO_SV))
            (STEO_idle)))
   (meets
    (SINGLE ((Gromit_Class Localize_Goal_SV))
            ((Localize_Goal_localize (?generation)))))
   ))


;(Define_Compatibility (SINGLE ((Gromit_Class STEO_SV))
;       ((STEO_read (?gen_loc ?status ?statusFlag))))
;  :duration_bounds [*latency* 3]
;  :compatibility_spec
;  (AND
;   (met_by
;   (SINGLE ((Gromit_Class SCorrel_SV))
;    ((SCorrel_scorrel (?gen_loc OK True))))
;   ))
;)

;;; Map

(Define_Compatibility  (SINGLE ((Gromit_Class Map_Goal_SV))
       ((Map_Goal_map (?generation))))
  :duration_bounds [*latency* _plus_infinity_]
  :compatibility_spec
  (AND
   (met_by
    (SINGLE ((Gromit_Class Map_Goal_SV))
    (Map_Goal_idle)))
   (met_by
     (SINGLE ((Gromit_Class Lane_SV))
     ((Lane_fuse (?generation OK True)))))
   (meets
    (SINGLE ((Gromit_Class Map_Goal_SV))
    (Map_Goal_idle)))
))
```

```
(Define_Compatibility  (SINGLE ((Gromit_Class Map_Goal_SV))
       (Map_Goal_idle))
  :duration_bounds [*latency* _plus_infinity_]
  :compatibility_spec
  (AND
   (met_by
    (SINGLE ((Gromit_Class Map_Goal_SV))
    ((Map_Goal_map (?_any_value_)))))
 )
)

(Define_Compatibility  (SINGLE ((Gromit_Class Lane_SV))
       ((Lane_fuse (?generation ?status ?statusFlag))))
  :duration_bounds [*latency* 5]
  :compatibility_spec
  (AND
   (met_by
   (SINGLE ((Gromit_Class Lane_SV))
   ((Lane_read (?generation OK True)))))
))

(Define_Compatibility  (SINGLE ((Gromit_Class Lane_SV))
       ((Lane_fuse (?generation OK True))))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class Lane_SV))
            (Lane_idle)))
   (meets
     (SINGLE ((Gromit_Class Map_Goal_SV))
    ((Map_Goal_map (?generation)))))
))

;(Define_Compatibility (SINGLE ((Gromit_Class Lane_SV))
;       ((Lane_read (?gen_loc ?status ?statusFlag))))
;  :duration_bounds [*latency* 3]
;  :compatibility_spec
;  (AND
;   (met_by
;    (SINGLE ((Gromit_Class SCorrel_SV))
;    ((SCorrel_scorrel (?gen_loc OK True))))
;   ))
;)

;;; shot_index > gen_loc

(Define_Compatibility (SINGLE ((Gromit_Class SCorrel_SV))
      ((SCorrel_scorrel (?shot_index ?status ?statusFlag))))
  :duration_bounds [*latency* 7]
  :compatibility_spec
  (AND
   (met_by
   (SINGLE ((Gromit_Class Camera_SV))
   ((Camera_shot (?shot_index OK True))))))
)

;;; Camera_SV
;;;
;;; Loop: Idle, Shot
;;; Shot when the poster is unused
;;;

;;; Camera_idle

(Define_Compatibility  (SINGLE ((Gromit_Class Camera_SV))
       ((Camera_idle (?prev ?shot_index))))
  :duration_bounds [*latency* _plus_infinity_]
)

(Define_Compatibility  (SINGLE ((Gromit_Class Camera_SV))
       ((Camera_idle (?prevgen ?shot_index))))
  :parameter_functions
  ((addeq(1 ?prevgen ?shot_index)))
  :compatibility_spec
  (AND
   (met_by
    (SINGLE ((Gromit_Class Camera_SV))
            ((Camera_shot (?prevgen ?_any_value_ ?_any_value_)))))
))

;;; Camera_shot

(Define_Compatibility  (SINGLE ((Gromit_Class Camera_SV))
       ((Camera_shot (?shot_index ?status ?statusFlag))))
  :duration_bounds [*latency* 3]
)

(Define_Compatibility  (SINGLE ((Gromit_Class Camera_SV))
       ((Camera_shot (?shot_index  ?status ?statusFlag))))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class Camera_SV))
            ((Camera_idle (?shot_index ?_any_value_)))))
   (met_by
    (SINGLE ((Gromit_Class Camera_SV))
            ((Camera_idle (?_any_value_ ?shot_index)))))
  (contained_by
    (SINGLE ((Gromit_Class CameraObs_SV))
            (CameraObs_idle)))
   (equal
    (SINGLE ((Gromit_Class Load_Goal_SV))
            ((Load_Goal_exec(?shot_index ?_any_value_)))))
   )
  )

;;; Scorrel
;;;
;;; Loop: Idle, Scorrel
;;; SCorrel meets Camera_Shot
;;; SCorrel contained-by Camera_Idle

;;; SCorrel_idle
(Define_Compatibility  (SINGLE ((Gromit_Class SCorrel_SV))
       ((SCorrel_idle(?shot_index))))
  :duration_bounds [*latency* _plus_infinity_]
)

(Define_Compatibility  (SINGLE ((Gromit_Class SCorrel_SV))
       ((SCorrel_idle(?shot_index))))
  :compatibility_spec
  (AND
;   (meets
;    (SINGLE ((Gromit_Class SCorrel_SV))
;            ((SCorrel_scorrel (?_any_value_)))))
   (met_by
    (SINGLE ((Gromit_Class SCorrel_SV))
            ((SCorrel_scorrel (?shot_index OK True)))))
))



;;; Scorrel_scorrel



(Define_Compatibility  (SINGLE ((Gromit_Class SCorrel_SV))
       ((SCorrel_scorrel (?shot_index ?status ?statusFlag))))
  :compatibility_spec
  (AND
   (contained_by
    (SINGLE ((Gromit_Class Camera_SV))
    ((Camera_idle (?_any_value_)))))
   (contained_by
    (SINGLE ((Gromit_Class CameraObs_SV))
    (CameraObs_idle )))
   (meets
    (SINGLE ((Gromit_Class SCorrel_SV))
    ((SCorrel_idle (?shot_index)))))
   (met_by
    (SINGLE ((Gromit_Class SCorrel_SV))
    ((SCorrel_idle(?_any_value_)))))
))

;;;; STEO
;;;;
;;;; loop: Idle, Read, Pos
;;;; Read meets SCorrel
;;;; Read contained-by STEO_Idle
;;;; Pos  meets Read

;;; STEO_idle

(Define_Compatibility  (SINGLE ((Gromit_Class STEO_SV))
       (STEO_idle))
  :duration_bounds [*latency* _plus_infinity_]
)

(Define_Compatibility  (SINGLE ((Gromit_Class STEO_SV))
       (STEO_idle))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class STEO_SV))
            ((STEO_read (?_any_value_))))))
)

;;; STEO_read
```

```
(Define_Compatibility  (SINGLE ((Gromit_Class STEO_SV))
       ((STEO_read (?shot_index  ?status ?statusFlag))))
  :duration_bounds [*latency* 3]
  )

(Define_Compatibility  (SINGLE ((Gromit_Class STEO_SV))
       ((STEO_read (?shot_index ?status ?statusFlag))))
  :compatibility_spec
  (AND
   (contained_by
    (SINGLE ((Gromit_Class Camera_SV))
    ((Camera_idle (?shot_index ?_any_value_)))))
   (contained_by
    (SINGLE ((Gromit_Class CameraObs_SV))
    (CameraObs_idle)))
   (contained_by
    (SINGLE ((Gromit_Class SCorrel_SV))
    ((SCorrel_idle(?shot_index))))))
)

(Define_Compatibility  (SINGLE ((Gromit_Class STEO_SV))
       ((STEO_read (?shot_index OK True))))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class STEO_SV))
    ((STEO_pos (?shot_index)))))
  )
)

;;; STEO_pos

;(Define_Compatibility  (SINGLE ((Gromit_Class STEO_SV))
;        ((STEO_pos (?status ?statusFlag ?scorrel_state))))
;  :duration_bounds [*latency* 8]
;  )


;;;; Lane
;;;;
;;;; loop: Idle, Read, Fuse
;;;; Read meets SCorrel
;;;; Read contained-by SCorrel_Idle
;;;; Pos  meets Read



;;; Lane_idle

(Define_Compatibility  (SINGLE ((Gromit_Class Lane_SV))
       (Lane_idle))
  :duration_bounds [*latency* _plus_infinity_]
)

(Define_Compatibility  (SINGLE ((Gromit_Class Lane_SV))
         (Lane_idle))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class Lane_SV))
    ((Lane_read (?_any_value_))))))
  )

;;; Lane_read


(Define_Compatibility  (SINGLE ((Gromit_Class Lane_SV))
       ((Lane_read (?shot_index ?status ?statusFlag))))
  :duration_bounds [*latency* 3]
  )

(Define_Compatibility  (SINGLE ((Gromit_Class Lane_SV))
       ((Lane_read (?shot_index OK True))))
  :compatibility_spec
  (AND
   (contained_by
    (SINGLE ((Gromit_Class SCorrel_SV))
    ((SCorrel_idle(?shot_index)))))
   (meets
    (SINGLE ((Gromit_Class Lane_SV))
    ((Lane_fuse(?_any_value_)))))
))

;;; Lane_fuse

(Define_Compatibility  (SINGLE ((Gromit_Class Lane_SV))
       ((Lane_fuse (?shot_index ?status ?statusFlag))))
  :duration_bounds [*latency* 5]
  )
```

```
;; (Define_Compatibility  (SINGLE ((Gromit_Class Global_Goal_SV))
;;         ((Global_Goal_exec (?gen ?ngen ?first))))
;;   :duration_bounds [*latency* _plus_infinity_]
;;   :parameter_functions
;;    ((addeq (1 ?gen ?ngen)))
;;   :compatibility_spec
;;   (?first
;;    OR
;;    (Single
;;     AND
;;      (met_by
;;       (SINGLE ((Gromit_Class Global_Goal_SV))
;;             ((Global_Goal_idle (?gen)))))
;;      (meets
;;       (SINGLE ((Gromit_Class Global_Goal_SV))
;;             ((Global_Goal_end (?ngen)))))
;;     )
;;    (First
;;     AND
;;      (met_by
;;       (SINGLE ((Gromit_Class Global_Goal_SV))
;;             ((Global_Goal_idle (?gen)))))
;;      (meets
;;       (SINGLE ((Gromit_Class Global_Goal_SV))
;;             ((Global_Goal_exec (?ngen ?_any_value_)))))
;;     )
;;    (Medium
;;     AND
;;      (met_by
;;       (SINGLE ((Gromit_Class Global_Goal_SV))
;;             ((Global_Goal_exec (?_any_value_ ?gen ?_any_value_)))))
;;      (meets
;;       (SINGLE ((Gromit_Class Global_Goal_SV))
;;             ((Global_Goal_exec (?ngen ?_any_value_ ?_any_value_)))))
;;     )
;;    (Last
;;     AND
;;      (met_by
;;       (SINGLE ((Gromit_Class Global_Goal_SV))
;;             ((Global_Goal_exec (?_any_value_ ?gen ?_any_value_)))))
;;      (meets
;;       (SINGLE ((Gromit_Class Global_Goal_SV))
;;             ((Global_Goal_end (?ngen)))))
;;     ; (meets
;;     ;  (SINGLE ((Gromit_Class Free_future_SV))
;;     ;        ((Free_future_now (?ngen ?_any_value_)))))
;;     )
;; ))

(Define_Compatibility  (SINGLE ((Gromit_Class Global_Goal_SV))
       ((Global_Goal_exec (?gen ?ngen ?first))))
  :duration_bounds [*latency* _plus_infinity_]
  :parameter_functions
   ((addeq (1 ?gen ?ngen)))
  :compatibility_spec
   (AND
     (met_by
      (SINGLE ((Gromit_Class Global_Goal_SV))
            ((Global_Goal_idle (?gen)))))
     (meets
      (SINGLE ((Gromit_Class Global_Goal_SV))
            ((Global_Goal_end (?ngen)))))
    )
)

(Define_Compatibility  (SINGLE ((Gromit_Class Global_Goal_SV))
       ((Global_Goal_exec (?gen ?ngen ?first))))
  :duration_bounds [*latency* _plus_infinity_]
  :compatibility_spec
   (AND
     (ends_after [0 0]
      (SINGLE ((Gromit_Class Localize_Goal_SV))
             ((Localize_Goal_localize (?gen)))))
     (ends_after [0 0]
      (SINGLE ((Gromit_Class Map_Goal_SV))
             ((Map_Goal_map  (?gen)))))
  )
)


(Define_Compatibility  (SINGLE ((Gromit_Class Global_Goal_SV))
       (Global_Goal_start))
  :duration_bounds [2 2]
)

(Define_Compatibility  (SINGLE ((Gromit_Class Global_Goal_SV))
       (Global_Goal_start))
  :compatibility_spec
  (AND
   (meets
     (SINGLE ((Gromit_Class Global_Goal_SV))
            ((Global_Goal_idle (1)))))
  )
)

;; Global_Goal_end visible at deliberative time only
(Define_Compatibility  (SINGLE ((Gromit_Class Global_Goal_SV))
       ((Global_Goal_end (?gen))))
  :duration_bounds [*latency* _plus_infinity_]
  :compatibility_spec
   (AND
     (met_by
      (SINGLE ((Gromit_Class Global_Goal_SV))
            ((Global_Goal_exec (?_any_value_ ?gen  ?_any_value_)))))
     (meets
      (SINGLE ((Gromit_Class Global_Goal_SV))
            ((Global_Goal_idle (?gen)))))
     ;(contains
     ; (SINGLE ((Gromit_Class Free_future_SV))
     ;       (Free_future_stretch)))
    ))


;; (Define_Compatibility  (SINGLE  ((Gromit_Class Global_Goal_SV))
;;  ((Global_Goal_idle2 (?gen ?ngen))))
;;   :duration_bounds [*latency* _plus_infinity_]
;;   :parameter_functions
;;   ((addeq(1 ?gen ?ngen)))
;;   :compatibility_spec
;;   (AND
;;    (meets (SINGLE ((Gromit_Class Global_Goal_SV))
;;    ((Global_Goal_exec (?ngen False)))))
;;    (met_by
;;     (SINGLE ((Gromit_Class Global_Goal_SV))
;;             ((Global_Goal_exec (?gen ?_any_value_)))))
;;    )
;; )

(Define_Compatibility
  (SINGLE ((Gromit_Class Global_Goal_SV))
  ((Global_Goal_idle (?gen))))
  :duration_bounds [*latency* _plus_infinity_]
;  :parameter_functions
;  ((addeq(*max_h_planning_time* ?_start_time_ ?start_horizon))
;   (addeq(4 ?start_horizon ?token_end))
;   (eq(?_end_time_ ?token_end)))
  :compatibility_spec
  (AND
   (met_by
    (SINGLE ((Gromit_Class Global_Goal_SV))
            ((Global_Goal_end (?gen)))))
;   (meets
;    (SINGLE ((Gromit_Class Global_Goal_SV))
;            ((Global_Goal_exec (?gen ?_any_value_ First)))))
   ;(contains [0 0] [0 0]
   ; (SINGLE ((Agent_Class Planner_SV))
;    ((Planning (?start_horizon *end_of_day* PLAN_TO_STANDBY)))))
;   (contained_by
;    (SINGLE ((Gromit_Class SCorrel_SV))
;            (SCorrel_idle)))
;   (contained_by
;    (SINGLE ((Gromit_Class Camera_SV))
;            ((Camera_idle (?_any_value_)))))

;   (contained_by
;    (SINGLE ((Gromit_Class Map_Goal_SV))
;            (Map_Goal_idle)))
;   (contained_by
;    (SINGLE ((Gromit_Class Localize_Goal_SV))
;            (Localize_Goal_idle)))

;   (contained_by
;    (SINGLE ((Gromit_Class STEO_SV))
;            (STEO_idle)))

;   (contained_by
;    (SINGLE ((Gromit_Class Lane_SV))
;            (Lane_idle)))
   ;(contained_by
   ; (SINGLE ((Gromit_Class Free_future_SV))
   ;         (Free_future_idle)))

  )
)

;; Goal-loader
(Define_Compatibility
  (SINGLE ((Gromit_Class Load_Goal_SV))
  ((Load_Goal_exec (?gen ?ngen ?status ?statusFlag))))
  :duration_bounds [*latency* _plus_infinity_]
```

```
  :parameter_functions
   ((addeq(*goal_cicle_step* ?gen ?ngen)))
  :compatibility_spec
   (AND
     (met_by
      (SINGLE ((Gromit_Class Load_Goal_SV))
            (Load_Goal_idle)))
     (meets
      (SINGLE ((Gromit_Class Load_Goal_SV))
            (Load_Goal_idle )))
    (equal
      (SINGLE ((Gromit_Class Camera_SV))
            ((Camera_shot(?gen ?_any_value_) ))))
;    (starts_after [0 0]
;      (SINGLE ((Gromit_Class Camera_SV))
;            ((Camera_shot(?_any_value_) ))))
   )
)

; The load goal token is forced to fail, and a new
; cicle can start
(Define_Compatibility  (SINGLE ((Gromit_Class Load_Goal_SV))
       ((Load_Goal_exec (?gen ?ngen ?status False))))
;  :duration_bounds [*latency* *latency*]
  :parameter_functions
   ((addeq(*goal_cicle_step* ?gen ?ngen)))
  :compatibility_spec
  (AND
;   (before [*latency* *latency*]
;     (SINGLE ((Gromit_Class Global_Goal_SV))
;            ((Global_Goal_idle (?_any_value_) ))))
  (before [5 400]
     (SINGLE ((Gromit_Class Global_Goal_SV))
           ((Global_Goal_end (?ngen) ))))
  )
)


(Define_Compatibility
  (SINGLE ((Gromit_Class Load_Goal_SV))
  (Load_Goal_idle))
  :duration_bounds [*latency* _plus_infinity_]
  :compatibility_spec
   (AND
     (met_by
      (SINGLE ((Gromit_Class Load_Goal_SV))
            ((Load_Goal_exec (?_any_value_)))))
   )
)

;;; Free the future ;;;; Free the future
;(Define_Compatibility  (SINGLE ((Gromit_Class Free_future_SV))
;        (Free_future_idle))
;  :duration_bounds [*latency* _plus_infinity_]
;  :compatibility_spec
;  (AND
;   (meets
;     (SINGLE ((Gromit_Class Free_future_SV))
;            ((Free_future_now (?_any_value_)))))
;   (met_by
;     (SINGLE ((Gromit_Class Free_future_SV))
;            (Free_future_stretch)))
;  )
;)


;(Define_Compatibility  (SINGLE ((Gromit_Class Free_future_SV))
;        (Free_future_stretch))
;  :duration_bounds [*latency* _plus_infinity_]
;  :compatibility_spec
;  (AND
;   (met_by
;     (SINGLE ((Gromit_Class Free_future_SV))
;            ((Free_future_now (?_any_value_)))))
;   (meets
;     (SINGLE ((Gromit_Class Free_future_SV))
;            (Free_future_idle)))
;  )
;)

;(Define_Compatibility  (SINGLE ((Gromit_Class Free_future_SV))
;        ((Free_future_now (?gen ?ngen ?status ?statusFlag))))
;  :duration_bounds [*latency* *latency*]
;  :parameter_functions
;   ((addeq(2 ?gen ?ngen)))
;  :compatibility_spec
;  (AND
;   (meets
;     (SINGLE ((Gromit_Class Free_future_SV))
;            (Free_future_stretch)))
;   (met_by
;     (SINGLE ((Gromit_Class Free_future_SV))
;            (Free_future_idle)))
;  ; (contained_by
;  ;   (SINGLE ((Gromit_Class Global_Goal_SV))
;  ;          ((Global_Goal_exec(?_any_value_ ?_any_value_ Last)))))

;  )
;)

; The free future token is forced to fail, the future is erased and a new
; cicle can start
;(Define_Compatibility  (SINGLE ((Gromit_Class Free_future_SV))
;        ((Free_future_now (?gen ?ngen ?status False))))
;  :duration_bounds [*latency* *latency*]
;  :parameter_functions
;   ((addeq(*goal_cicle_step* ?gen ?ngen)))
;  :compatibility_spec
;  (AND
;   (before [*latency* *latency*]
;     (SINGLE ((Gromit_Class Global_Goal_SV))
;            ((Global_Goal_idle (?_any_value_) ))))
;  (before [40 200]
;     (SINGLE ((Gromit_Class Global_Goal_SV))
;            ((Global_Goal_end (?ngen) ))))
;  )
;)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;; AWAY_SV
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;; Away_monitor
;(Define_Compatibility  (SINGLE ((Gromit_Class Away_SV))
;        (Away_monitor ))
;  :duration_bounds [*latency* 4]
;)

;;meets Away_pos
;(Define_Compatibility  (SINGLE ((Gromit_Class Away_SV))
;        (Away_monitor ))
;  :compatibility_spec
;  (AND
;   (meets
;    (SINGLE ((Gromit_Class Away_SV))
;            ((Away_pos(?_any_value_) ))))
;   (meets
;    (SINGLE ((Gromit_Class Photo_obj_goal_SV))
;            (Photo_obj_goal_task)))
;  ))

;;;;;;;;;;;;;; Away_pos
(Define_Compatibility  (SINGLE ((Gromit_Class Away_SV))
       ((Away_pos (?xpos ?xFlag ?status ?statusFlag))))
  :duration_bounds [*latency*  _plus_infinity_]
)

;;;meets Away_buff
(Define_Compatibility  (SINGLE ((Gromit_Class Away_SV))
       ((Away_pos (?xpos ?xFlag OK True))))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class  Away_SV))
             ((Away_buff (?xpos))))))
)

;;;meets Rflex_speed_cntl
(Define_Compatibility  (SINGLE ((Gromit_Class Away_SV))
       ((Away_pos (?xpos ?xFlag OK True))))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class  Rflex_speed_cntl_SV))
             ((Rflex_cntl (OFF  ?_any_value_))))))
)

;;;;;;;;;;;;;;;;;;;;;; Away_buff
(Define_Compatibility  (SINGLE ((Gromit_Class Away_SV))
       ((Away_buff (?xbuf))))
  :duration_bounds [*latency*  _plus_infinity_]
)

;;;meets platine_pos
(Define_Compatibility  (SINGLE ((Gromit_Class Away_SV))
       ((Away_buff (?xbuf))))
  :compatibility_spec
  (AND
```

```
   (meets
    (SINGLE ((Gromit_Class  Platine_SV))
             ((Platine_pos (?xbuf ?_any_value_)))))
   (met_by
    (SINGLE ((Gromit_Class  Away_SV))
             ((Away_pos (?xbuf True OK True)))))
   (ends_during ;to be modified
    (SINGLE ((Gromit_Class  Rflex_speed_cntl_SV))
             ((Rflex_speed_cntl_idle (OFF)))))
  )
)

;;;meets away idle
(Define_Compatibility  (SINGLE ((Gromit_Class Away_SV))
       ((Away_buff (?xbuf))))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class Away_SV))
            (Away_idle )))
   (meets
    (SINGLE ((Gromit_Class Platine_SV))
            ((Platine_pos(?xbuf ?_any_value_) ))))
  )
)

;;;;;;;;;;;;;;;;;;;;;; Away_idle
(Define_Compatibility  (SINGLE ((Gromit_Class Away_SV))
       (Away_idle))
  :duration_bounds [*latency* _plus_infinity_]
)

;;meets Away_pos
(Define_Compatibility  (SINGLE ((Gromit_Class Away_SV))
       (Away_idle))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class Away_SV))
            ((Away_pos (?_any_value_))))))
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Rflex_speed_cntl_SV
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;; Rflex_cntl
(Define_Compatibility  (SINGLE ((Gromit_Class Rflex_speed_cntl_SV))
       (( Rflex_cntl(?ycntl ?status ?statusFlag) )))
  :duration_bounds [*latency* 4]
)

;;;meets Away_pos if ON and camera shot localize
(Define_Compatibility  (SINGLE ((Gromit_Class Rflex_speed_cntl_SV))
       ((Rflex_cntl(?ycntl ?status ?statusFlag))))
  :compatibility_spec
  (?ycntl
   OR
   (ON
    AND
    (meets
     (SINGLE ((Gromit_Class  Away_SV))
      ((Away_pos(?_any_value_) ) )) )
    (met_by
     (SINGLE ((Gromit_Class Rflex_speed_cntl_SV))
             ((Rflex_speed_cntl_idle (OFF)))) )
    (met_by
     (SINGLE ((Gromit_Class Platine_SV))
             ((Platine_back (OK True)))) )
  ; (ends_before [0 0]
  ;   (SINGLE ((Gromit_Class  CameraObs_SV))
  ;           (CameraObs_shotted )))
     )
   (OFF
    AND
    (met_by
     (SINGLE ((Gromit_Class  Away_SV))
             ((Away_pos (?_any_value_ True  OK True)))) )
    (met_by
     (SINGLE ((Gromit_Class Rflex_speed_cntl_SV))
             ((Rflex_speed_cntl_idle (ON)))) )
   (before [0 400] ;added for deliberative version (to be removed for reactive version)
    (SINGLE ((Gromit_Class  Platine_SV))
             ((Platine_pos (?_any_value_)))) )
  )
 )
)

;;;meets idle
```

```
(Define_Compatibility  (SINGLE ((Gromit_Class Rflex_speed_cntl_SV))
       ((Rflex_cntl(?ycntl OK True))))
  :compatibility_spec
   (AND
     (meets
        (SINGLE ((Gromit_Class Rflex_speed_cntl_SV))
                ((Rflex_speed_cntl_idle (?ycntl)))))
   )
)

;;;

;;;;;;;;;;;;;;;Rflex_speed_cntl_idle
(Define_Compatibility  (SINGLE ((Gromit_Class  Rflex_speed_cntl_SV))
                               ((Rflex_speed_cntl_idle (?rcntl))))

  :duration_bounds [*latency* _plus_infinity_]
)

(Define_Compatibility
  (SINGLE ((Gromit_Class Rflex_speed_cntl_SV))
          ((Rflex_speed_cntl_idle (?rcntl))))
  :compatibility_spec
   (AND
     (meets (SINGLE ((Gromit_Class Rflex_speed_cntl_SV))
                    ((Rflex_cntl (?_any_value_)))) )
     (met_by (SINGLE ((Gromit_Class Rflex_speed_cntl_SV))
                    ((Rflex_cntl (?rcntl)))) )
   )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Platine_SV
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;; Platine_Idle

(Define_Compatibility  (SINGLE ((Gromit_Class Platine_SV))
       (Platine_idle))
  :duration_bounds [*latency* _plus_infinity_]
)

(Define_Compatibility  (SINGLE ((Gromit_Class Platine_SV))
       (Platine_idle ))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class Platine_SV))
           ((Platine_pos(?_any_value_))))) )
)

;;; Platine_oriented
(Define_Compatibility  (SINGLE ((Gromit_Class Platine_SV))
       (Platine_oriented))
  :duration_bounds [*latency* _plus_infinity_]
)

;;; Platine_oriented
(Define_Compatibility  (SINGLE ((Gromit_Class Platine_SV))
       (Platine_oriented ))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class Platine_SV))
           ((Platine_back(?_any_value_)))))
   ;(met_by ;PROBLEMA
    ;(SINGLE ((Gromit_Class Platine_SV))
     ;       ((Platine_pos(?_any_value_)))))
   )
)

;;;;;;;;;; Platine_Pos
(Define_Compatibility  (SINGLE ((Gromit_Class Platine_SV))
       ((Platine_pos (?ypos ?status ?statusFlag))))
  :duration_bounds [*latency* 10]
  )

;;;;;meets platine_oriented
(Define_Compatibility  (SINGLE ((Gromit_Class Platine_SV))
       ((Platine_pos (?ypos OK True))))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class Platine_SV))
           (Platine_oriented)))
   (contained_by
    (SINGLE ((Gromit_Class Rflex_speed_cntl_SV))
           ((Rflex_speed_cntl_idle(OFF)))))
   (contained_by ;overconstraint
    (SINGLE ((Gromit_Class SCorrel_SV))
           ((SCorrel_idle(?_any_value_)))))
   (contained_by ;overconstraint
    (SINGLE ((Gromit_Class STEO_SV))
           (STEO_idle)))
   (contained_by
    (SINGLE ((Gromit_Class Camera_SV))
           ((Camera_idle(?_any_value_)))))
  )
)

(Define_Compatibility  (SINGLE ((Gromit_Class Platine_SV))
       ((Platine_pos (?ypos  ?status ?statusFlag))))
  :compatibility_spec
   (AND
    (met_by
      (SINGLE ((Gromit_Class Away_SV))
           ((Away_buff(?ypos))))) )
)


;;;;; meets Camera_shot
(Define_Compatibility  (SINGLE ((Gromit_Class Platine_SV))
       ((Platine_pos (?ypos OK True))))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class  CameraObs_SV))
            ((CameraObs_shot (?_any_value_))))))
)

;;; Platine_back

(Define_Compatibility  (SINGLE ((Gromit_Class Platine_SV))
       ((Platine_back(?status ?statusFlag))))
  :duration_bounds [*latency* 10]
)

;;;;;meets idle
(Define_Compatibility  (SINGLE ((Gromit_Class Platine_SV))
       ((Platine_back (OK True))))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class Platine_SV))
           (Platine_idle)))
   (meets
    (SINGLE ((Gromit_Class  Rflex_speed_cntl_SV))
            ((Rflex_cntl (ON ?_any_value_)))))
   (met_by
    (SINGLE ((Gromit_Class  CameraObs_SV))
            ((CameraObs_shot (OK True)))))
  )
)
;;;;; met_by Away_buff
;(Define_Compatibility  (SINGLE ((Gromit_Class Platine_SV))
;       ((Platine_pos (?ypos OK True))))
;  :compatibility_spec
;  (AND
;   (met_by
;    (SINGLE ((Gromit_Class  Away_SV))
;            ((Away_buff (?ypos))))))
;)


;;;;; Photo_Obj_Goal_SV
;(Define_Compatibility  (SINGLE ((Gromit_Class Photo_obj_goal_SV))
;       (Photo_obj_goal_task))
;  :duration_bounds [*latency* 43000]
;)

(Define_Compatibility  (SINGLE ((Gromit_Class Photo_obj_goal_SV))
       (Photo_obj_goal_task))
  :duration_bounds [*latency* _plus_infinity_]
  ;:parameter_functions
  ;;((addeq(?_end_time_ *eod_buffer* *end_of_day*)))
  ;;((?_end_time_ <- Calculate_End_Time (?_start_time ?eod_buffer ?eod)))
   ;((leq(?_end_time_ *max_goal_task_time*)))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class Photo_obj_goal_SV))
           (Photo_obj_goal_end)))
   (ends_before [0 0]
    (SINGLE ((Gromit_Class  Rflex_speed_cntl_SV))
            ((Rflex_cntl (ON OK True)))))

   ;(starts_during [-4 4] [4 _plus_infinity_]
   ; (SINGLE ((Agent_Class Planner_SV))
;    (Planner_Idle)))

  )
)

(Define_Compatibility  (SINGLE ((Gromit_Class Photo_obj_goal_SV))
       (Photo_obj_goal_end))
  :duration_bounds [*latency* _plus_infinity_]
)

(Define_Compatibility  (SINGLE ((Gromit_Class Photo_obj_goal_SV))
       (Photo_obj_goal_end))
  :compatibility_spec
  (AND
   ;(meets
   ; (SINGLE ((Gromit_Class Photo_obj_goal_SV))
   ;         (Photo_obj_goal_idle)))
   (met_by
    (SINGLE ((Gromit_Class Photo_obj_goal_SV))
            (Photo_obj_goal_task)))
  )
)

(Define_Compatibility  (SINGLE ((Gromit_Class Photo_obj_goal_SV))
       (Photo_obj_goal_start))
  :duration_bounds [8 _plus_infinity_]
)

(Define_Compatibility  (SINGLE ((Gromit_Class Photo_obj_goal_SV))
       (Photo_obj_goal_start))
  :compatibility_spec
  (AND
   (meets
     (SINGLE ((Gromit_Class Photo_obj_goal_SV))
            (Photo_obj_goal_idle)))
   ;(met_by
   ; (SINGLE ((Gromit_Class Photo_obj_goal_SV))
   ;         (Photo_obj_goal_task)))
  )
)

;(Define_Compatibility  (SINGLE ((Gromit_Class Photo_obj_goal_SV))
;        ((Photo_obj_goal_idle (?start_horizon ?token_end) )))
;  :duration_bounds [32 _plus_infinity_]
;)


(Define_Compatibility
  (SINGLE ((Gromit_Class Photo_obj_goal_SV))
  (Photo_obj_goal_idle ))
  :duration_bounds [*latency* _plus_infinity_]
  ;:parameter_functions
  ;((addeq(*max_h_planning_time* ?_start_time_ ?start_horizon))
  ; (addeq(4 ?start_horizon ?token_end))
  ; (eq(?_end_time_ ?token_end)))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class Photo_obj_goal_SV))
            (Photo_obj_goal_task)))
  )
)

;;;;;;; CameraObs_SV
;;; CameraObs_shot

(Define_Compatibility  (SINGLE ((Gromit_Class CameraObs_SV))
       ((CameraObs_shot (?status ?statusFlag))))
  :duration_bounds [*latency* 3]
)

(Define_Compatibility  (SINGLE ((Gromit_Class CameraObs_SV))
       ((CameraObs_shot (?status ?statusFlag))))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class CameraObs_SV))
            (CameraObs_shotted )))
   (met_by
    (SINGLE ((Gromit_Class CameraObs_SV))
            (CameraObs_idle )))
   (contained_by
        (SINGLE ((Gromit_Class Platine_SV))
               (Platine_oriented)))
   (meets
    (SINGLE ((Gromit_Class  Platine_SV))
             ((Platine_back (?_any_value_)))))
   (contained_by
        (SINGLE ((Gromit_Class Camera_SV))
               ((Camera_idle (?_any_value_)))))
    )
  )
```

```
;;; Camera_idle

(Define_Compatibility  (SINGLE ((Gromit_Class CameraObs_SV))
       (CameraObs_idle ))
  :duration_bounds [*latency* _plus_infinity_]
)

(Define_Compatibility  (SINGLE ((Gromit_Class CameraObs_SV))
       (CameraObs_idle ))
  :compatibility_spec
  (AND
   (meets
    (SINGLE ((Gromit_Class CameraObs_SV))
            ((CameraObs_shot (?_any_value_)))))
  )
)

;;; Camera_shotted
(Define_Compatibility  (SINGLE ((Gromit_Class CameraObs_SV))
       (CameraObs_shotted ))
  :duration_bounds [*latency* _plus_infinity_]
)


;meets idle
(Define_Compatibility  (SINGLE ((Gromit_Class CameraObs_SV))
       (CameraObs_shotted ))
  :compatibility_spec
  (AND
   (met_by
    (SINGLE ((Gromit_Class CameraObs_SV))
               ((CameraObs_shot (?_any_value_)) )) )
   (meets
    (SINGLE ((Gromit_Class CameraObs_SV))
               (CameraObs_idle  )) )
   (ends_before [0 0]
     (SINGLE ((Gromit_Class Rflex_speed_cntl_SV))
              ((Rflex_cntl(ON OK True))) )
  )
)
)

;EOF
```