Reasoning About Robot Actions: A Model Checking Approach

Khaled Ben Lamine and Froduald Kabanza

Dept. de Math-Info
Université de Sherbrooke
Sherbrooke, Québec J1K 2R1, Canada
{benlamin,kabanza}@dmi.usherb.ca

Abstract. Mobile robot control remains a difficult challenge in changing and unpredictable environments. Reacting to unanticipated events, interacting and coordinating with other agents, and acquiring information about the world remain difficult problems. These actions should be the direct products of the robot's capabilities to perceive, act, and process information intelligently, taking into account its state, that of the environment, and the goals to be achieved. This paper discusses the use of model-checking to reason about robot actions in this context. The approach proposed is to study behaviors that allow abstract, but informative models, so that a computer program can reason with them efficiently. Model-checking can then be used as a means for verifying and planning robot actions with respect to such behaviors.

1 Introduction

Mobile robot control remains a difficult challenge in unstructured or dynamic environments in which operating conditions are changing and unpredictable. In such instances, the behavior-based robot programming paradigm advocates distributing a robot task among several processes (called behaviors), running concurrently, each dealing with a simpler subtask [1]. A behavior operates by updating one or more robot control parameters at different steps of execution. For instance, a behavior to avoid an obstacle is implemented by a program that changes the robot's heading and speed, at every step of execution, depending on the robot's sensors. A behavior that moves the robot to a target position controls the same parameters, but based on target coordinates instead of the robot's current position. In principle, the latter behavior is not supposed to account for obstacles. Rather it should be the combination of both behaviors that move the robot to the target position while avoiding obstacle. Different behavior-based architectures have been proposed, all agreeing on the concurrent nature of behaviors. They differ in mechanisms used to implement this concurrency and abstraction [2, 3]

Designing behaviors as concurrent processes facilitates their design, but introduces, at the same time, all the fundamental problems of concurrent processes. One typical problem is deadlocks, which appear when the execution of a process

is blocked by a condition that was supposed to be enabled by another process. In some cases, deadlocks are normal and intended in the design. For example, an obstacle-avoidance process is blocked while waiting for an obstacle to be signaled by a separate obstacle-detection process. In other cases, deadlocks are not intended at all and they represent design faults that should be fixed. To illustrate, because of programmer oversight, the obstacle-detection process fails to signal a detected obstacle in the obstacle-avoidance process. Another typical problem is livelocks, which appear when the concurrent execution fails to progress towards a desired execution point, for instance, because of pathological cyclic execution by one or more processes. These kinds of design errors are very difficult to detect, yet they can cause fatal robot behaviors.

Similar problems have been being investigated in the area of protocol verification, where successful tools have been developed to automatically detect typical error designs, using model-checking techniques [4]. The basic idea is to simulate protocol rules (or their models) in such a way that the simulation covers all their potential executions, and to check that every simulated execution sequence satisfies a correctness statement. If the statement is specified as a formula in temporal logic, then the problem becomes to check that the execution sequence satisfies the formula; in other words, we check that the execution sequence is a logical model of the formula, that is, model checking. One important problem in this approach is developing efficient modeling and simulation techniques to cover the entire space of execution sequences for given protocol rules. Lesser coverage will yield proportional confidence in the correctness of the protocol rules, but not complete confidence. It may nevertheless be useful as a debugging tool.

We propose adopting similar ideas to improve the design of robot behaviors. As with automatic protocol verification, we would like to check whether the concurrent execution of robot behaviors satisfies a given correctness property. There are three contexts to which we would like to apply this: off-line verification, online verification, and planning. For off-line verification, we would like to simulate a correctness condition to be verified on a set of behaviors, covering the space of their possible executions and checking that they satisfy the correctness condition. For online verification, given a progress condition that should be satisfied by a normal execution of robot behaviors, we would like to track the execution, checking that it does not violate the progress condition. Goals represent tasks for behaviors and may be changed at any time by user requests or by other trigger events. Changes in goals require a reconfiguration of behaviors. Reconfiguration may also be required when the environment context changes, for example, upon failure detection. Behavior planning addresses these reconfiguration problems. Given a goal statement and an environment context, we would like to simulate given behaviors to select combinations or configurations that best suit a given context or goal.

If robot behaviors were like network protocol rules, we could easily solve the above problems through the straightforward adaptation of tools tailored to verify protocol rules. Unfortunately, there are important fundamental differences between protocol rules and robot behaviors. Robot actions (e.g., turn,

stop, accelerate, grasp an object or release it) rely on noisy sensors and error-prone actuators. Although noise also exists in protocols (because of unreliable transmission media), it is more easily abstracted over by checking for corrupted packets at lower levels. Consequently, the problem of verifying protocols deals with higher level transmission rules, such as ensuring logical consistency in the acknowledgement of received packets [4]. Another important difference between robot behaviors and communication protocols is that robot behaviors are more clearly goal-oriented. Hence, the notion of a "goal" should be part of the language used to express correctness statements, so that we can check, for example, if two behaviors involve conflicting goals.

In this paper we discuss some steps relating to the development of more suitable tools for verifying robot behaviors, whether in the context of online verification, off-line verification, or planning. We propose using Linear Temporal Logic (LTL) [5] as the language for specifying properties of robot behaviors. As explained above, depending on the context, such a property will express a design correctness statement, an execution progress condition, or a goal. As these ideas are at the early stage of experimentation, the discussion in this paper remains at a relatively abstract level.

The remainder of the paper is organized as follows. The next section briefly discusses coding robot behaviors using the SAPHIRA control architecture. Section 3 deals with specifying properties of behaviors using LTL. Section 4 discusses a technique for efficiently checking that a behavior trace violates an LTL property and the application of this technique to monitor real-time executions, to detect off-line design errors, and to plan behaviors. We conclude with a discussion on related work.

2 Coding Robot Behaviors

2.1 Robot Platform

We use one indoor Pioneer-1 mobile robot and one outdoor all-terrain Pioneer-1.[1] Figure 1 shows a snapshot of both robots in our lab. Each robot is equipped with 8 sonars (allowing to detect obstacles), a pan-tilt-zoom color camera, an onboard image processing system (allowing recognition of 3 different colors simultaneously), a radio modem (that transmits information from the robot to a remote computer and vice versa), an audio-video transmitter that transmits the camera video output to the computer, and grippers. Information transmitted from the robot to the remote computer via the radio modem mainly consists of sonar readings, robot motor and wheel status, features extracted by the onboard image processor, and the gripper status. Information transmitted from the remote computer to the robot consists of commands on actuators. There are few basic robots control parameters that directly affect the actuators: heading angle (i.e., turning angle for the wheels), speed, camera configuration (pan, tilt, zoom) and gripper configuration (open, closed).

[1] Pioneer-1 mobile robot is a trademark of ActivMedia Inc.

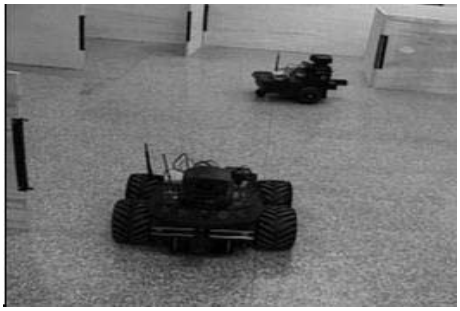

Fig. 1. Pioneer-1 and Pioneer-1 AT robots

2.2 SAPHIRA Behaviors

We program robot behaviors using SAPHIRA architecture [2], which is based on a synchronous model of concurrency. Each behavior is launched with a given priority. Behaviors are executed by a process scheduler, which cycles through them every 100 milliseconds. At every cycle, the scheduler considers each active behavior to determine its output control actions; actions from all processes are combined to determine their joint effect on the robot's actuators, using behavior priorities to resolve conflicts. How control actions are joined cannot be fully described here, but it can be approximated by saying that when behaviors with different priorities affect the same control variable with conflicting values, then the behavior with the highest priority takes precedence. If the behaviors have the same priority, then their conflicting effects on the control variables are merged. For example, if one indicates turning left 45 degrees and the other turning right with 45 degrees, the end result will be moving front (i.e., turn 0 degrees).

To illustrate, let us consider oversimplified specifications of some of the basic robot navigation behaviors that are provided with SAPHIRA (Figure 2). For the sake of concision, we have removed variable declarations that are not crucial for understanding the examples, but that are required for a complete and correct definition. Obstacle-avoidance is implemented by two behaviors: `avoidCollision` (which avoids obstacles close to the robot) and `keepOff` (which veers the robot away from longer obstacles). Both behaviors are similar in description, but react differently as one focuses on close obstacles while the other is detecting long distant obstacles. Behavior `goToPos` moves the robot to a given position. The command `Turn Left get_turn_amount` turns the robot's wheel to the left the number of degrees indicated by `get_turn_amount`. This variable is updated periodically using rules that are declared in the behavior (omitted here for the sake of concision) but roughly the amount is proportional to how far to the left or right the obstacle is. The command `Turn Right` is similar. The command `Speed` sets the robot's speed to a given value in mm/sec.

Variables are local to behaviors and have a fuzzy interpretation. To simplify the examples here, we assume a binary interpretation for variables in the rule an-

```
 BeginBehavior avoidCollision
  If obstacle_right Then Turn Left  turning_depth
  If obstacle_left  Then Turn Right turning_depth
  If obstacle_front Then Turn preferred_turn turning_depth
 EndBehavior
BeginBehavior keepOff
  If obstacle_right Then Turn Left  turning_depth
  If obstacle_left  Then Turn Right turning_depth
  If obstacle_front Then Turn preferred_turn turning_depth
EndBehavior
BeginBehavior goToPoS
  If gt_too_fast Or gt_too_slow Then Speed gt_speed
  If gt_pos_on_left  Then Turn Left  gt_turn_amount
  If gt_pos_on_right Then Turn Right gt_turn_amount
  If gt_pos_achieved Then Speed 0.0
EndBehavior
```

Fig. 2. Avoid-collision, keep-off and go-to behaviors

tecedents and crisp values for command `Turn` and `Speed`. In `avoidCollision`, `obstacle_left` is set to true when the sensor readings indicate a close obstacle to the left of the robot (say within a range of 2 meters). In `keepoOff`, `obstacle_left` is set to true when the sensor readings indicate there is an obstacle to the left of the robot but further away. An analogue interpretation holds for the other variables. The reader is referred to the SAPHIRA programming manual for a more accurate description of behaviors, but the above abstract examples are sufficient to illustrate our point.

For our examples, the basic control variables affected are robot heading (through command `Turn`) and speed. Consider the situation depicted in Figure 3(a): the robot's goal is to reach the position marked by the large black dot, from a position marked by the large white dot. Presently, there are no obstacles, either close or distant. In this instance, `avoidCollision` and `keepOff` have no effect on the control variables; `goToPos` changes the robot's heading to align it with the target position. In Figure 3(b), the robot has detected a distant obstacle in front, but no close obstacle. In this situation, `keepOff` veers the robot to the right to avoid the obstacle. In Figure 3(c), the robot has detected a distant obstacle in front as well as closer obstacles on the left and right, resulting in conflicting actions. `keepOff` would have the robot veer to the right or left; `avoidCollision` and `goToPos` would have it move front. The SAPHIRA operating system joins the actions, yielding the action of moving straight ahead.

The above behaviors implement obstacle avoidance while moving towards the target position with obstacles. This works quite well in most cases. Yet it is not difficult to find a configuration of obstacles that causes the robot to oscillate between two positions without ever making progress towards the goal (e.g., see Figure 3(d)). In such situations, the robot needs some global path planning. It would, however, run counter to the principle of behavior-based approaches

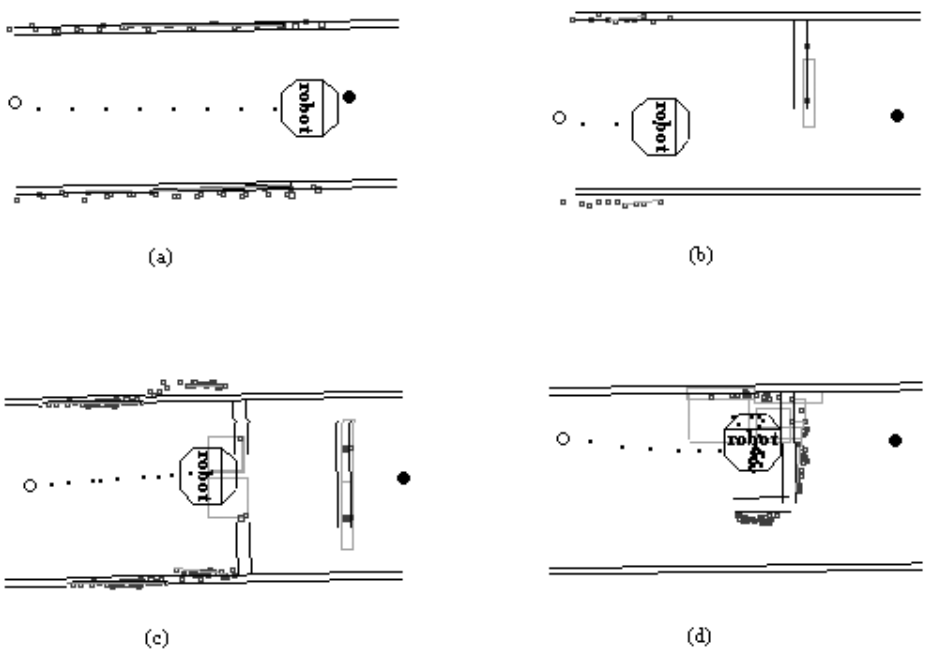

Fig. 3. Robot navigation snapshots

to include a path planner in the obstacle-avoidance behavior itself. Complex tasks should result from the combination of simple behaviors, not some complex deliberative process.

In keeping with the spirit of behaviors, it would be better to use tools that can monitor the robot's behaviors in order to detect their failures. Such tools can enable us to keep our basic navigation behaviors with the knowledge that they fit some contexts but may fail in some tricky situations. In the latter, a recovery behavior can be invoked to put the robot on the right track. In other words, what is needed is a way to specify progress conditions under which behaviors or combination of behaviors are deemed to be progressing normally. Then, we can have another behavior or process that monitors those progress conditions to send signals when they are violated. That way, with the previous example, we would write a progress condition stating that "as long as the robot has not reached the target position, then its position should progress towards the goal more than 10 m every 60 s." This approach is general and goes beyond navigation behaviors. We would also use it, for example, to control a robot grasping objects by writing progress statement such as "the robot keeps approaching the object only as long as the object is in the visual field of the camera." With higher-level object delivery behaviors, we may say something like "the robot remains within a region until some condition holds, such as until another robot puts an object there required by the first robot."

3 Specifying Properties of Behaviors

A robot state is described by specifying control variables and their respective values. Basic state properties are expressed using predicates, that is, functions over states, that return `true` or `false`, depending on whether or not the property holds. For instance, the predicate `=(x,y)` is defined to return `true` if `x` is equal to `y`. As this example shows, the state argument is implicit in the notation of a predicate. With an appropriate definition, the predicate `in(x,y)` evaluates to `true` in a state in which object `x` or robot `x` is in room `y`. Predicates can have function applications as arguments. For instance, the statement `=(speed(robot),v)` evaluates to `true` in a state in which the robot's speed is `v mm/sec`.

First-order-logic is a predicate language allowing to write more complex logical expressions (called formulas) from predicates by using the logic connectives `&&` (and), `||` (or) , `!` (not), `->` (implies), `forall` and `exists`. Our use of the `forall` quantifier is always bounded by a predicate, that is, we can only quantify on something true of a given a predicate. For instance, we can write

```
forall (x) in(x,Lab3031)
   (<=(position-x,30) &&  >=(position-x,0) &&
    <=(position-y,30) && >=(position-y,0))
```

The predicate `in(x,Lab3031)` here is mandatory. This means for every `x` such that `x` is in `Lab3031`, the position of `x` satisfies the indicated boundary constraints.

First-order logic allows us to write statements that are true about a given state. In order to write statements that are true about behaviors, we need a logic that can express statements about possible execution sequences. LTL is such a language obtained from First-order logic by introducing operators that are applied to formulas to relate them to the future of a given execution sequence or to its past.[2]

3.1 Formal Syntax

The LTL formula formation rules are:

1. a predicate is the most simple LTL formula (including the primitive propositions `true` and `false`);
2. for any LTL formulas `f` and `g`, and time interval `i`, then the following are also LTL formulas:

[2] First-order logic becomes suitable for expressing properties of execution sequences if we include a state argument in predicates. For example, "in(x,Lab3031,s)" would mean that "in(x,Lab3031)" holds in state "s". This extension of First-order logic is known as Situation-Calculus [6]. LTL relates predicates to states in a different way using modal temporal operators, leading to a different approach for evaluating formulas over execution sequences.

(a) `! f` (intuitively: "not `f`"),
(b) `f && g` (intuitively: "`f` and `g`"),
(c) `next i f` (intuitively: "the next state is on the time interval `i` and satisfies `f`");
(d) `f until i g` (intuitively: "on the forwards time interval `i`, `f` holds in every future state up to a state satisfying `g`");
(e) `last i f` (intuitively: "the last state is on the time interval `i` and satisfies `f`");
(f) `f since i g` (intuitively: "`f` holds in every past state up to a state satisfying `g`, on the time interval `i`")
(g) `forall x1 ... xn p f` (intuitively: "for all `x1 ... xn` such that `p` holds, then `f` holds", where `p` is a predicate involving `x1 ... xn` as free variables, and `f` is a formula also possibly involving those free variables);
(h) Parenthesis may be used to resolve ambiguities.

The future (or forwards) operators (`next` and `until`) refer to future of an execution sequence, that is, the subsequence rooted from the current state. The past (or backwards) operators (`last` and `since`) are like their mirror reflections referring to the history of an execution sequence, that is, the subsequence starting from initial state for the execution sequence and ending in the current state. The arguments of a temporal operator are a time interval and one or two formulas. The time interval is noted `[i,j]`, where `i` is the starting time, assuming the time in the current state is 0, and `j` is the ending time. The ending time is noted `?` when it is infinite. This interpretation holds going forwards if the interval is associated to a future operator, or going backwards if it is associated to a past operator. In our case, the time unit is generally set to the length of a cycle for the SAPHIRA operating system (i.e., 100 milliseconds.) The following abbreviations are standard:

- `f || g` is equivalent to `!(!f && !g)` (intuitively: "`f` or `g`").
- `f->g` is equivalent to `!f || g` (intuitively: "`f` implies `g`")
- `eventually i f` is equivalent to `true until g` (intuitively: "`f` eventually holds in some future state on the time interval `i`").
- `always i f` is equivalent to `! eventually i !f` (intuitively: "`f` holds in every future state on the time interval `i`").
- `previously i f` is equivalent to `true since i f` (intuitively: "`f` holds on some past state on the time interval `i`").
- `alwaysPreviously i f` is equivalent to `! previously i !f` (intuitively: "`f` holds in past future state on the time interval `i`").
- `exists x1 ... xn p f` is equivalent to `! forall x1 ... xn p !f` (intuitively: "there exists `x1, ..., xn` such that if `p` holds then `f` holds");

3.2 Examples of LTL formulas

1. The LTL formula `always (0,?) !(active(B1) && active(B2))` states that behavior `B1` and behavior `B2` must never be active at the same time.

2. The formula

```
always [0,?] (! in(robot,Lab3031) ->
              next [0,?] (in(robot,Lab3031) ->
                          eventually [0,2]
                           always [0,100] active(B1)))
```

   means that, once the robot enters Lab3031, then behavior `B1` must be active within 2 time units and remain active during 100 time units.
3. The formula

```
always [0,?] (forall (x) in(x,Lab3031)
               (grasping(robot,x) -> requested(x)))
```

   means that, in Lab3031, the robot should only grasp requested object.
4. The formula

```
always [0,?]
 (((last [0,?] ! in(robot,Lab3031)) &&
   in(robot,Lab3031)) ->
  eventually [0,2] (always [0,100] active(B1)))
```

   is just another way of expressing the same statement as in (2) above.
5. The formula

```
always [0,?] ((alwaysPreviously [0,10)] stalled()) ->
              next [0,?] active(B3))
```

   means that, if the robot is stalled since 10 time units, then behavior `B3` must be active in the next state.
6. The formula

```
always [0,?]((last [0,?] clost() &&
              last [0,?] last [0,?] clost() &&
              last [0,?] last [0,?] last [0,?] clost()) ->
             next [0,?] active(B4))
```

   means that, if we have three consecutive losses of communication with the robot (`clost`), then behavior `B4` must be active in the next state.

3.3 Formal Semantics

Reactive behaviors, such as obstacle-avoidance, are cyclic by nature, with no predefined terminating point. Such a cyclic execution unwinds into an infinite execution sequence. Thus the semantics of LTL formulas is defined by considering infinite execution sequences. In this case, a terminating execution sequence is represented by an equivalent one obtained by replicating the terminal state infinitely.

The LTL interpretation rules are recursively defined in Figure 4. This function takes three arguments, respectively, an LTL formula `h`, an execution sequence `E`, and a state `s` on `E`. It returns `true` if `h` holds in `s`; otherwise, it returns `false`. This must be only regarded as a specification of the semantics rule, not as an effective procedure for checking LTL formulas over an execution sequence. Since we assume an infinite execution sequence, the "algorithm" does not actually terminate. In fact, we realized that it facilitates understanding when the semantics rules are given that way in an algorithmic style. Below, we explain simple modifications to this specification to obtain an effective method for checking LTL formulas.

The basic case is with predicates. The generic function `holdPredicate` takes a predicate and state as arguments and then calls a domain-dependent predicate-evaluation function that returns `true` if the predicates holds in the state, `false` otherwise.

The recursive case is domain-independent and implements the interpretation of logical connectives and temporal operators. A formula `! f` holds on `s` if `f` does not hold on `s`. A conjunctive formula holds on `s` if each conjunct holds on `s`.

A formula of the form `next i f` holds on `s` if `f` holds on the successor of `s`, and this successor appears in the interval `i`; note that every state has a successor since the sequence is infinite. Function `succ(s,E)` returns the state immediately following `s` on the execution sequence `E`; `dur(s,s1,E)` returns the duration between state `s` and state `s1` on the execution sequence `E`; `lb(i)` returns the lower bound of a time interval `i`; `ub(i)` returns the upper bound.

The interpretation rule for `until` means `f until i g` is satisfied in the current state if the interval `i` is active (i.e., its lower bound is 0) and `g` is satisfied in the current state, otherwise if `f` is satisfied in the current state and `(f until j g)` is satisfied in the next state, with `j` representing a reduction of `i` with the time elapsed between the current and the next state. For a situation in which the upper-bound of an interval is `?`, `?-d` reduces to `?` for any duration `d`.

The interpretation rules for `previously` and `since` are analogue to, respectively, `next` and `until`, but it goes backwards, using the function `pred(s,E)` to access to the state before `s` on the execution sequence `E` (this yields `null` if `s` is the initial state).

Finally, `forall x1 ... xn p f` holds in a state if `f` holds in that state for every instantiation of `x1, ..., xn` that makes `p` true in the state.

4 Monitoring and Planning Behaviors

4.1 Monitoring Robot Behaviors

The SAPHIRA operating system cycles through behaviors every 100 ms to determine their effects on robot control parameters, resulting in a state change every 100 ms. However, abstract properties do not require such a fine granular level of state sampling. For instance, if we are checking whether or not a robot enters a

```
hold(h,E,s) {
 if (h is a predicate) return holdPredicate(h,s);
 if (h is of the form (! f)) return (! hold(f,E,s));
 if (h is of the form (f && g)) return (hold(f,E,s) && hold(g,E,s));
 if h is of the form (next i f)
    return ((lb(i) <= dur(s,succ(s,E),E)) &&
            (ub(i) >= dur(s,succ(s,E),E)) &&
            hold(f,succ(E,s),E));
 if (is of the form (f until i g)) {
    let s1 = succ(E,s);
    let j  = [max(0,lb(i) - dur(s,s1,E)), max(0,ub(i) - dur(s,s1,E))];
    if (j == [0,0]) return hold(g,E,s);
    else return ((lb(i)==0 && ub(i) != 0 && hold(g,E,s)) ||
                 (hold(f,E,s) && hold((f until j g),E,s1)));}
 if (h is of the form (last i f))
     return ((pred(s) == void) ||
             ((lb(i) <= dur(pred(s,E),s,E)) &&
              (ub(i) >= dur(pred(s,E),s,E)) &&
              hold(f,E,pred(s))));
 if (is of the form (f since i g)) {
    let s1 = pred(E,s);
    if (s != null)
       let j = [max(0,lb(i) - dur(s1,s,E)), max(0,ub(i) - dur(s1,s,E))];
    else let j = [0,0];
    if (j == [0,0]) return hold(g,E,s);
    else return ((lb(i)==0 && ub(i) != 0 && hold(g,E,s)) ||
                 (hold(f,E,s) && hold((f since j g),E,s1)));}
 if (is of the form (forall x1 ... xn p f)) {
    let result = true;
    for every p1 obtained from p by instantiating (x1, ..., xn)
    such that holdPredicate(p1,s) {
      let f1 obtained from f by applying the same instantiation;
      result = result && hold(f1,E,S);}
    return result;}}
```

Fig. 4. LTL Semantics

given room, we may check this at a periodicity in terms of minutes rather than milliseconds. For this, we define *state-sampling frequency* parameter, at which states are sampled. This depends on the type of progress conditions being monitored. Not all robot control variables have to be tracked. Only those relevant to atomic propositions in LTL progress conditions need to be involved. For this, we introduce a function *state-reflector* that tracks relevant state features.[3]

By keeping a trace of the robot state during an execution, we check whether or not the robot's execution up to the current point of execution does not violate given LTL progress conditions (for example those in Section 3.2). However, rather than explicitly applying the LTL interpretations, we use a more efficient approach by checking the LTL progress conditions *on the fly*, keeping information relevant to the history in past formulas rather than explicitly in trace, and delaying the evaluation of future formulas in the next state. This technique is known as "progressing LTL formulas over a sequence of states". It was first introduced for future operators, in the TLPLAN planning system [7]. Here we extend it to formulas with past operators.

4.2 Progressing Formulas

The idea is to keep track of the following: (1) the *current state*, (2) a trace of execution to the current state and (3) a set of *delayed LTL formulas*, each corresponding to a "delayed" progress condition in the current state. Each of these is updated at every execution cycle (i.e., at every state change).

Intuitively, a delayed formula is one which would have been evaluated in the next successor state by a recursive call to `hold`, but instead, had its evaluation postponed. The delayed formula is also called "progressed formula", because it is progressed from one state to another.

Initially, the current state is the initial state of the robot (as given by the *state reflector*); the trace is empty; the set of delayed formulas is the given set of progress conditions.

At a current state, this is how each of the three components are updated. The new state, successor of the current one, is automatically given by the *state reflector*. The new trace is obtained from the current one by appending the current state.[4] For each LTL progress condition, a corresponding new LTL formula is obtained by invoking a "delayed" evaluation of the LTL-interpretation procedure on the current formula, current trace and current state, assuming the new state as successor and a time duration between them equal to the *state-sampling period*.

[3] The SAPHIRA operating system maintains a basic robot's control state in a structure accessible to user-written programs. This includes the current robot position, speed, heading, and sonar readings. The status of behaviors (active or suspended) is also accessible to user-written programs and hence can be traced.

[4] This is a naive approach for storing traces. We are investigating more efficient approaches, where we can exploit syntactic information about the LTL progress conditions being monitored to only store partial, but sufficient information about the trace.

By "delayed" evaluation of LTL-interpretation procedure, we mean an invocation of the procedure in Figure 4, such that each recursive call to `hold` involving the new state as argument just returns its input formula without further evaluation; that is, the evaluation is delayed by just returning the formula. All other recursive calls are made, leading either to `true` or `false`. With simplifications, the final result is either `true`, `false` or a formula whose main connective is `next`. This is the new, updated LTL progression condition.

That way, the formula `f'` returned by the delayed evaluation of `f` expresses would have to be satisfied by any execution sequence starting from the new state in order for a corresponding execution just one step earlier in the current state to satisfy `f`. A proof of this claim simply follows from the fact that this is a delayed evaluation of the LTL semantics rules.

Hence, if the delayed formula is `true` this means that any future execution is satisfactory; we can stop monitoring the corresponding progress condition since it becomes valid from this point of execution. On the other hand, if the delayed formula is `false`, this means that we have a violation of the progress condition; we must thus send a notification to a handling behavior.

If the delayed formula is other than `true` or `false`, this means that so far the execution is progressing normally. There may still be conditions requiring eventual achievement in the future, but none of them has been violated so far.

It is interesting to note that a formula like `(eventually (0, ?) p)`, for any proposition `p`, is never violated by an execution trace. As long as the current trace does not contain a state satisfying `p`, this will be progressed to `(eventually (0, ?) p)`. Intuitively, as long as we have not encountered `p`, we still have a chance of meeting such a state later. It is only at the end of the execution sequence that we can conclude that `p` is not satisfied. A trace that does not contain a state satisfying `p` does not violate `(eventually (0, ?) p)`, but it does not satisfy it either.

In contrast, if we have the formula `(eventually (0, 10) p)`, then at every step, this will be progressed into a similar formula, but with the upper bound of the time interval decremented by the duration between the current and next state. Soon or later, we will reach a state in which 10 time units have elapsed from the initial state, yet without having a met a state satisfying `p` (if we met a state satisfying `p` the delayed formula would have been `true`). This will be progressed to `false`, thus indicating a failure of the progress condition.

Properties that are only violated by infinite executions (or cyclic executions in practice) are usually called liveness properties [5]. They are conveyed by `until` or `eventually` operators with an unbound time-intervals. The opposite are safety properties, which are conveyed by `until` or `eventually` operators with bounded-time intervals, and `always` operators regardless of their time-intervals. In fact, an `until` operator with an unbounded-time actually conveys both a safety property (because its first formula component must be maintained `true`) and a liveness property (because its second component must be eventually made `true`, without a deadline).

4.3 Planning Robot Behaviors

Monitoring is about checking that runtime traces do not violate progress conditions. Planning deals with checking that predicted, simulated future behaviors would continue satisfying those progress conditions. That way, the robot can anticipate failures to re-configure its behaviors.

While a runtime execution is deterministic, a simulation involves nondeterministic guesses. This is so because the effects and preconditions of actions in future states depend, in part, on real-time conditions unknown in the current state.[5] Hence, when simulating behaviors, we must account for all possibilities; which yields a simulation graph rather than a single execution sequence.

In order to simulate behaviors efficiently, we need a *state transition model* that hides details of behaviors that are irrelevant to progress conditions being monitored. The state-transition model specifies preconditions for the execution of a robot's actions and their effects. Specifying it is one of the most challenging hurdles in our approach. Currently, it is specified independently using STRIPS or ADL action descriptions [7], but we are investigating approaches that would make it possible to ground the state-transition model in the source code of SAPHIRA behaviors.

The simulation is essentially a search process through the space of possible future executions of robot behaviors at a level of abstraction described by the state-transition model. In this context, it is helpful to classify progress conditions into goals (i.e., tasks that the robot must achieve) and search control strategies that guide the search process [7]. LTL search-control formulas can be understood as additional constraints on the behaviors reflecting behaviors that are irrelevant to some goals and, hence, that should not be simulated.

Liveness properties pose an additional difficulty when planning for cyclic behaviours. A property like `(eventually (0, ?) p)` expresses a goal of eventually achieving `p`, without a deadline. Formula-progression alone cannot check the violation of such properties, unless some simplifying assumptions are made, such as fixing one goal state [7], or approximating goals without deadlines with goals with arbitrary deadlines [8]. [6] Cyclic behaviours can be dealt with by explicitly checking that goals of achievement without deadline are not violated by cyclic behaviours, but this introduces an additional complexity [10].

[5] Since SAPHIRA uses a synchronous model of concurrency, no nondeterminism is entailed in the interleaving of processes as in the asynchronous case. With UNIX processes, for example, we would have to guess how the UNIX operating system would interleave them.

[6] This is the approach taken by Drummond and Bresina, although using a goal language that is a subset of LTL and a model-checking procedure less general than formula-progression [8]. In general, assuming that only propositions are negated, using goals with deadlines limit formulas to only safety properties and hence completely verifiable by using formula-progression alone [9].

4.4 Summary

The key software components composing our approach are summarized in Figure 5. The *state extractor* implements an interface between SAPHIRA and LTL, by extracting state features that are relevant to predicates appearing in the LTL specification of progress conditions. The *Runtime monitor* checks given progress conditions over runtime behavior. This is done incrementally, by labelling each state of the trace with the corresponding LTL formulas, using LTL formula progression. A formula progressed to `false` in a given state means that the corresponding condition is violated by the trace. For each violated formula, a signal is generated indicating which formula is violated and at what point of the trace. The signal is then handled by an ad-hoc behavior.

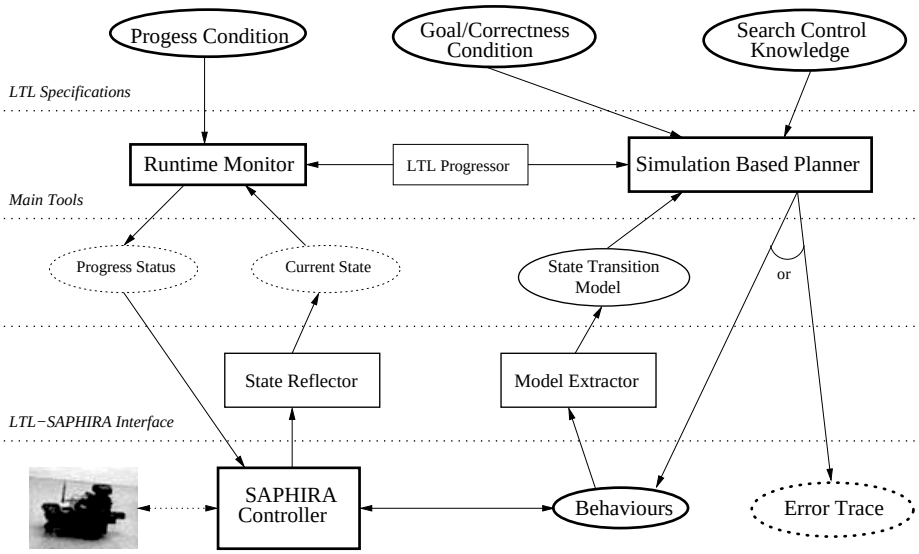

Fig. 5. Software Architecture

On the other hand, from a state-transition model, we simulate execution sequences from a given context; a simulation-based planner evaluates LTL goal statements over those sequences, allowing to determine combinations of behaviors whose future execution would be expected to best satisfy the goal, that is, planned behaviors. In a different mode, the planner accepts correctness criterion to validate over simulated behaviors.

As the implementation currently stands, the monitor and the planner are operational, but both still need validation on more realistic applications. This is where most of the current implementation effort is being driven. Besides the automatic extraction of state-transition models from SAPHIRA behaviors that

is mentioned above, we are also working on a more efficient approach for storing past information during a simulation. This is essential when planning is done with a goal that is a mixture of past and future operators, because a trace has to be stored somehow in every state. As most states share traces, the question is to represent this efficiently into the planner. This is less problematic in monitoring because there is always one single trace.

5 Conclusion

Pure behavior-based approaches are simply reactive. They involve no explicit representation of the robot's goals, plans, or internal "world model." Goals are only implicit in the situation-action coupling and plans emerge as one action is executed in a way that it triggers or deactivates another. This is done in a modular way, in which simple behaviors are run concurrently to achieve complex behaviors. This simplicity facilitates the design of immediate real-time responses to sensed events. This is well illustrated by the remarkable performance of behavior-based approaches in robot navigation, using very simply coded behaviors. On the other hand, deliberative approaches use explicit representation of the world model constructed from sensory data and the actions executed by the robot emerge from the interplay of an explicit planner (or other formal reasoning component) on the world model and given robot goals. One clear advantage of this approach is that behaviors can be planned automatically. Having both features in a robot control architecture leads to hybrid control architectures.

Our motivation is not to propose yet another hybrid architecture. Instead, we are proposing tools to both monitor and plan robot behaviors using the same basic technique, that is, checking that execution sequences generated from runtime or simulated behaviors satisfy LTL correctness statements. Experimenting with this approach in the SAPHIRA architecture facilitates its implementation because SAPHIRA is already a deliberative architecture allowing symbolic representation of the robot world model. In particular, all basic feature of the robot's control state are available symbolically, such as robot speed, heading, and activation status of processes. Hence, it is easy to define LTL propositions on top of these features and other user-defined control variables.

Our approach can also be integrated with other behavior-based architectures, provided it is possible to extract symbolic state information. Schönherr et al. describe a method doing that for connected behavior-based architectures [11]. More precisely, their approach makes it possible to extract symbolic facts characterizing the activation of behaviors. These facts could be considered as proposition from which LTL progress conditions could be defined, which enable the use of our formal monitoring tool. In a similar vein, but with respect to planning, Nicolescu and Mataric discuss an idea about relating behaviors (coded in Ayllu architecture) to STRIPS-like operators [12]. This is compatible with our approach since it makes it possible to simulate Ayllu behaviors through the application of STRIPS operators to obtain state sequences for LTL goal checking.

In the earlier stage of this work, we started with a fuzzy version of LTL [13]. This was motivated by the use of fuzzy control in SAPHIRA. In the end, we introduced past operators and planning, and at the same time abstracted over fuzzy-control, so as to keep uniformity in the language used for planning and monitoring. Haslum is also exploring techniques similar to ours for monitoring and predicting control systems for unmanned aerial vehicles [14]. Our approach can also be related, to a limited extent, to a research program being conducted by Alur *et al.* [15]. They are experimenting the use of automata-theoretic methods to synthesize robot behaviors that are constructively proven to satisfy some logical properties.

6 Acknowledgement

This work is supported by the Canadian Natural Sciences and Engineering Research Council (NSERC). We also would like to thank the reviewers for helpful comments.

References

1. Arkin, C.: Behavior-Based Robotics. The MIT Press (1998)
2. Konolige, K., Myers, K., Ruspini, E., Saffiotti, A.: The Saphira architecture: A design for autonomy. Journal of Experimental and Theoretical Artificial Intelligence **9** (1997) 215–235
3. Werger, B.: Ayllu: Distributed port-arbitrated behavior-based control. In: Proc. of 5th International Symposium on Distributed Autonomous Robotic Systems (DARS). (2000) 24–35
4. Holzmann, G.: Design and Validation of Computer Protocols. Prentice-Hall (1991)
5. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag (1991)
6. De Giacomo, G., Lespérance, Y., Levesques, H.: Congolog, a concurrent programming language based on the situation calculus. Artificial Intelligence **121** (2000) 109–169
7. Bacchus, F., Kabanza, F.: Using temporal logics to express search control knowledge for planning. Artificial Intelligence **116** (2000) 123–191
8. Drummond, M., Bresina, J.: Anytime synthetic projection: Maximizing probability of goal satisfaction. In: Proc. of 8th National Conference on Artificial Intelligence (AAAI 90), MIT Press (1990) 138–144
9. Barbeau, M., Kabanza, F., St-Denis, R.: Synthesizing plant controllers using real-time goals. In: Proc. of 14th International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann (1995) 791–798
10. Kabanza, F., Barbeau, M., St-Denis, R.: Planning control rules for reactive agents. Artificial Intelligence **95** (1997) 67–113
11. Schönherr, F., Cistelecan, M., Hertzberg, J., Christaller, T.: Extracting situation facts from activation value histories in behavior-based robots. In: Proc. of Joint German/Austrian Conference on AI, LNAI Vol. 2174. (2001) 305–319
12. Nicolescu, M., Mataric, M.: Extending behaviour-based systems capabilities using an abstract behaviour representation. In: Proc. of AAAI Fall Symposium on Parallel Cognition. (2000) 27–34

13. Ben Lamine, K., Kabanza, F.: History checking of temporal fuzzy logic formulas for monitoring behavior-based mobile robots. In: Proc. of the 12th IEEE International Conference on Tools with Artificial Intelligence. (2000) 312–319
14. Haslum, P.: Models for prediction. In: In IJCAI Workshop on Planning under Uncertainty and Incomplete Information. (2001) 8–17
15. Alur, R., *et al.*: A framework and architecture for multirobot coordination. In: Proc. 7th International Symposium on Experimental Robotics. (2001) 289–300