Under consideration for publication in Formal Aspects of Computing

Reasoning about Goal-Directed Real-Time Teleo-Reactive Programs

Brijesh Dongol[1,2], Ian J. Hayes[2] and Peter J. Robinson[2]

[1]Department of Computer Science, The University of Sheffield, S1 4DP, UK
[2]School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Australia

Abstract. The teleo-reactive programming model is a high-level approach to developing real-time systems that supports hierarchical composition and durative actions. The model is different from frameworks such as action systems, timed automata and TLA$^+$, and allows programs to be more compact and descriptive of their intended behaviour. Teleo-reactive programs are particularly useful for implementing controllers for autonomous agents that must react robustly to their dynamically changing environments. In this paper, we develop a real-time logic that is based on Duration Calculus and use this logic to formalise the semantics of teleo-reactive programs. We develop rely/guarantee rules that facilitate reasoning about a program and its environment in a compositional manner. We present several theorems for simplifying proofs of teleo-reactive programs and present a partially mechanised method for proving progress properties of goal-directed agents.

Keywords: Teleo-reactive programming, goal-directed agents, rely/guarantee reasoning, real-time programs, reactive systems, interval-based logics

1. Introduction

Software is increasingly being used to implement controllers for safety-critical applications in real-time environments [**?**, **?**, **?**]. For such systems, failures can have a high cost, and hence it is important to ensure dependability of the underlying software. As the applications become more sophisticated, the programming languages and the logics that one uses must accordingly become more sophisticated. This paper is concerned with methods for specifying and proving correctness of *goal-directed agents*, which are agents that progress towards a main goal by achieving a number of intermediate subgoals. The environments of such agents are assumed to be dynamic and may hinder the agent from achieving any of its goals, and may even cause the agent to re-establish previously achieved goals.

The teleo-reactive programming language, developed by Nils Nilsson, is a high-level language that has been shown to be useful for implementing controllers for autonomous agents that react robustly to continually changing environments [**?**, **?**, **?**, **?**]. We present a logic over dense time intervals to formalise the real-time semantics of teleo-reactive programs. The logic combines aspects of Duration Calculus [**?**] and temporal logic [**?**, **?**]. To facilitate compositional reasoning, we develop rely/guarantee-style reasoning rules [**?**], where the *rely* condition describes properties of the environment and the *guarantee* condition describes how the program will behave under the assumption that the rely

Correspondence and offprint requests to: B.Dongol@sheffield.ac.uk, {Ian.Hayes,pjr}@itee.uq.edu.au

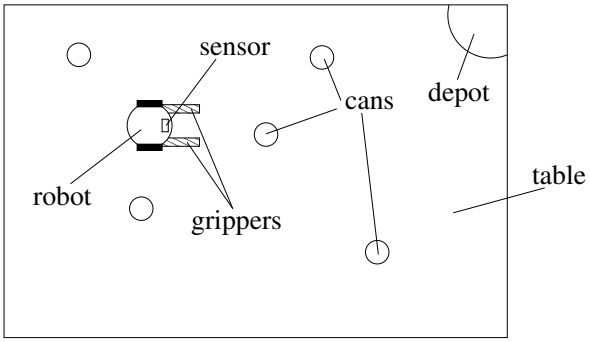

Fig. 1. Top down view of can clearing robot example

condition holds. Our framework allows one to verify guarantee properties that hold over all intervals (to prove safety properties), intervals of a certain length (to prove progress properties), and about the lengths of the intervals (to prove timing properties). We provide a number of proof rules for simplifying proofs of progress in goal-directed agents. In particular, we focus on a progression theorem that allows a rely condition that is sufficient for a program to satisfy progress to be derived.

In this paper, we introduce teleo-reactive programs by presenting a controller for a can-collecting robot in Section **??**. Furthermore, we present:

- an interval-based real-time temporal logic (Section **??**),
- formalisation of the semantics of teleo-reactive programs (Section **??**) using the logic in Section **??**,
- a compositional rely/guarantee-style theory for proving properties of teleo-reactive programs with a focus on controllers for goal-directed agents (Section **??** and Section **??**),
- a method of decomposing proof of progress properties by automatically generating the necessary proof obligations and assumptions (Section **??**).

In Section **??**, we demonstrate our approach by proving that the can-clearing robot example in Fig. **??** achieves its goal of moving cans from the table to the depot. As part of the proof, we develop the required rely conditions, i.e., assumptions on the environment that are necessary for the robot to make progress.

2. Teleo-reactive programs

We explain teleo-reactive programs by considering the program in Fig. **??**, which implements a controller for the robot depicted in Fig. **??**. The robot must clear cans from the table by moving them to the depot. The robot is able to sense when it is pointing towards a can or the depot, rotate on its axis (to scan the environment for cans or the depot), move forward (in the direction of its front sensor), and grasp/ungrasp its grippers (to pick up and drop cans). The primitive actions nil, grasp, ungrasp, rotate and forward from Fig. **??** are formalised in Example **??**.

The main program robot specifies the set of output variables as $rot.\mathring{robot}$, $pos.\mathring{robot}$ and $\mathring{gdist}$, where $\mathring{x}$ denotes the rate of change of variable x (c.f. [**?**]). Thus, the outputs of the robot modify the rate of change of the robot rotation, robot position and distance between the grippers, respectively. For each guarded program $c \rightarrow \mathsf{M}$, c is a state predicate and M is either a (primitive) durative action or a sequence of guarded programs. The guard c must hold continuously over any interval over which M is executed. In a sequence of guarded programs, the guards that appear earlier in a sequence are given priority over those that appear later. For example, in a sequence $\langle c_1 \rightarrow \mathsf{M}_1, c_2 \rightarrow \mathsf{M}_2 \rangle$,

- if the guard c_1 ever becomes true, then M_2 stops and M_1 begins executing,
- if c_1 ever becomes false, then
  - M_2 is executed if c_2 holds, and
  - if neither c_1 nor c_2 holds, then neither M_1 nor M_2 is executed and the behaviour is chaotic [**?**].

Hence, the guard of M_2 is effectively $\neg c_1 \wedge c_2$. Note that the effective guards must hold continuously over the intervals in which M_1 and M_2 execute. Each of the programs in Fig. **??** avoid chaotic behaviour because the last guard of each program is $true$. Program robot executes so that:

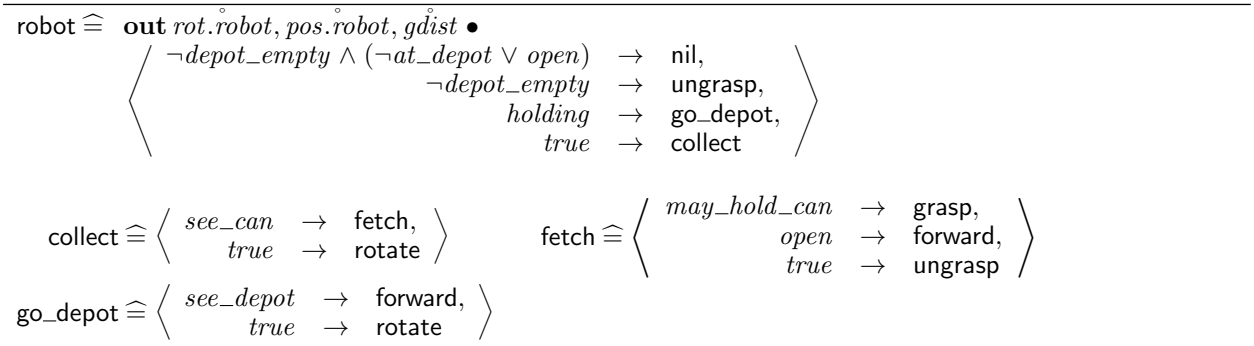

Fig. 2. Teleo-reactive controller for the can-clearing robot

- while there is already a can in the depot (i.e., $\neg depot\_empty$ holds), and if the robot is at the depot the grippers are fully open (i.e., $at\_depot \Rightarrow open$ holds), the robot executes action nil (which does nothing),
- else, while the depot is non-empty the robot executes ungrasp to release the can that it may currently be holding,
- else, while the robot is holding a can, the robot delivers the can it is holding to the depot by executing program deliver,
- else, the robot attempts to collect a can by executing collect.

Because negations of earlier guards appear as conjuncts to each guard, the effective guard of ungrasp within program robot implicitly implies $at\_depot \wedge \neg open$.

Teleo-reactive programs are often structured so that the overall goal of the agent is represented by the guard of the first action of the program, i.e., executing the first action represents achievement of the goal. Hence, the first action of such a teleo-reactive program is often nil to indicate that the goal has been reached. The rest of the actions must establish subgoals corresponding to one of the guards of the actions earlier in the list.

Teleo-reactive programs are useful for implementing controllers for goal-directed agents in dynamically changing environments [**?**]. For the robot program, the environment of the robot may add/remove cans from the table, the depot and the robot's grasp. The environment may also move cans around the table. Thus, the behaviour of the environment may both help and hinder the robot from achieving its task. The robot program reacts to both circumstances and switches to (different) subgoals in order to progress towards the overall goal of moving a can to the depot. For example, given that the depot stays empty during the execution of go_depot, it may be possible for $holding$ to become false (e.g., due to an environment action), in which case the robot switches to program collect. Conversely, if $holding$ becomes true during the execution of collect, the robot switches to program go_depot, which causes the robot to move the collected can to the depot and to deposit it there. Note that even after achieving the overall goal, the environment may change in a way that causes the agent to attempt re-establishing the goal. For example, in the robot program, the robot will attempt to collect and deliver a can to the depot if the depot is empty. After a can has been placed in the depot, the environment may remove the can from the depot (e.g., via a conveyor belt or a chute) which causes the robot to begin searching for a can to move from the table to the depot once again.

Teleo-reactive programs support hierarchical composition (nesting) in a straightforward manner. For example, program collect (which itself expands into a program fetch and an action rotate) is nested hierarchically within robot. Although collect is a self-contained program, within the context of the robot program, collect only executes over intervals in which $depot\_empty \wedge \neg holding$ is continuously true. Hence, the guard of fetch is effectively $depot\_empty \wedge \neg holding \wedge see\_can$, and similarly the guard of rotate is effectively $depot\_empty \wedge \neg holding \wedge \neg see\_can$. Our proof rules exploit this structuring and allow properties of the subprograms nested within a higher-level program to be used in proofs of properties of higher-level programs.

To see the benefits of teleo-reactive programming, we compare the program in Fig. **??** with its hybrid action system [**?**] equivalent in Fig. **??**, where variables $speed$ and rot represent the forward speed and rotation of the robot, respectively and $gripper$ represents the movement of the grippers. Note that the implicit negations of guards and updates to variables in Fig. **??** must be made explicit, which results in more complicated guards, and hence a more complicated program. The purpose of each action is difficult to understand and the goal-directed structure of the teleo-reactive program is lost. Furthermore, we are not able to reason about the properties of each component separately and combine these to prove properties of the whole system. Unlike action systems where the state is modified by assigning

do	$\neg depot\_empty \wedge (\neg at\_depot \vee open)$	$\rightarrow$	$speed, rot, gripper := 0, 0, idle$	(nil)
[]	$\neg depot\_empty \wedge at\_depot \wedge \neg open$	$\rightarrow$	$speed, rot, gripper := 0, 0, release$	(ungrasp)
[]	$depot\_empty \wedge holding \wedge see\_depot$	$\rightarrow$	$speed, rot, gripper := \xi, 0, idle$	(forward)
[]	$depot\_empty \wedge holding \wedge \neg see\_depot$	$\rightarrow$	$speed, rot, gripper := 0, \kappa, idle$	(rotate)
[]	$depot\_empty \wedge \neg holding \wedge may\_hold\_can$	$\rightarrow$	$speed, rot, gripper := 0, 0, close$	(grasp)
[]	$depot\_empty \wedge see\_can \wedge \neg may\_hold\_can \wedge open$	$\rightarrow$	$speed, rot, gripper := \xi, 0, idle$	(forward)
[]	$depot\_empty \wedge see\_can \wedge \neg may\_hold\_can \wedge \neg open$	$\rightarrow$	$speed, rot, gripper := 0, 0, release$	(ungrasp)
[]	$depot\_empty \wedge \neg see\_can$	$\rightarrow$	$speed, rot, gripper := 0, \kappa, idle$	(rotate)
od				

Fig. 3. Hybrid action system equivalent of program in Fig. **??**

to variables, teleo-reactive programs modify the state by executing *durative actions* (see Example **??**). Hence, there may be several consecutive iterations of an action system without a state change, which is avoided in a teleo-reactive program. A teleo-reactive program is able to switch from any program M_i to another M_j if the actual (explicit) guard of M_i becomes $false$ and the actual guard of M_j becomes $true$.

3. A temporal logic for intervals

In this section we present the logic for reasoning about teleo-reactive programs that we have developed. Because the actions of a teleo-reactive program are durative, Linear Temporal Logic [**?**, **?**], which is defined for discrete traces of states is inappropriate. An interval temporal logic for discrete traces [**?**] has been extended to dense streams to obtain the Duration Calculus [**?**]. However, Duration Calculus assumes all intervals are closed, allows adjoining intervals to overlap and uses the *almost everywhere* operator. A state predicate holds almost everywhere in an interval iff it is only false for a set of measure zero. This is inappropriate for our purposes because we aim to extend our work with time bands [**?**] where a single time at one time scale may expand to an interval of time at another time scale. Thus, we develop a logic that is influenced by interval temporal logic [**?**] and Duration Calculus [**?**] but is better suited to later incorporation with the theory of time bands [**?**, **?**, **?**].

In this logic we consider the behaviour of teleo-reactive programs on time intervals. Furthermore, we restrict time intervals to be of finite length. We can do this because although teleo-reactive programs can theoretically run forever, any machine on which we run a program has only a finite lifetime. Hence we consider the behaviour of teleo-reactive programs on arbitrary finite length time intervals as defined below. Curiously, infinite intervals are often used to do termination proofs for procedural code in order to distinguish terminating and non-terminating executions. Because teleo-reactive programs are non-terminating by nature, infinite intervals do not need to be considered.

3.1. Intervals

We let $Time \mathrel{\widehat{=}} \mathbb{R}$ denote the set of all times. We consider an interval to be a contiguous finite-length non-empty subset of $Time$ and allow intervals to be open/closed at either end. Given that $(l, u) \mathrel{\widehat{=}} \{t{:}\ Time \mid l < t < u\}$ and $[l, u] \mathrel{\widehat{=}} \{t{:}\ Time \mid l \leq t \leq u\}$ respectively denote open and closed intervals from $l \in Time$ to $u \in Time$, an interval has type

$$Interval \mathrel{\widehat{=}} \{\Delta \subseteq Time \mid \Delta \neq \{\} \wedge \exists l, u \in Time \bullet (l, u) \subseteq \Delta \subseteq [l, u])\}.$$

We use '.' for function application and let $\mathsf{glb}.\Delta$ and $\mathsf{lub}.\Delta$ denote the greatest lower and least upper bounds of interval Δ, respectively. The *length* of an interval Δ is given by

$$\ell.\Delta \mathrel{\widehat{=}} \mathsf{lub}.\Delta - \mathsf{glb}.\Delta.$$

Thus, each interval Δ includes all times between $\mathsf{glb}.\Delta$ and $\mathsf{lub}.\Delta$ and may or may not include these bounds.

For intervals Δ and Δ', we say Δ *adjoins* Δ' (denoted $\Delta \propto \Delta'$) iff the following holds.

$$\Delta \propto \Delta' \mathrel{\widehat{=}} (\mathsf{lub}.\Delta = \mathsf{glb}.\Delta') \wedge (\Delta \cup \Delta' \in Interval) \wedge (\Delta \cap \Delta' = \{\})$$

That is, $\Delta \propto \Delta'$ states that Δ' is an interval that immediately follows Δ. The sets of prefixes and suffixes of Δ are

defined as follows.

$$\mathit{prefix}.\Delta \mathrel{\widehat{=}} \{\Delta' \in \mathit{Interval} \mid \Delta' \subseteq \Delta \wedge \forall t \in \Delta, t' \in \Delta' \bullet t \leq t' \Rightarrow t \in \Delta'\}$$
$$\mathit{suffix}.\Delta \mathrel{\widehat{=}} \{\Delta' \in \mathit{Interval} \mid \Delta' \subseteq \Delta \wedge \forall t \in \Delta, t' \in \Delta' \bullet t \geq t' \Rightarrow t \in \Delta'\}$$

3.2. Interval predicates

We assume that variable names are taken from the set Var. A *state space* is given by $\Sigma_V \mathrel{\widehat{=}} V \to \mathit{Val}$, which is a total function from a set of variable names, $V \subseteq \mathit{Var}$ to a set of values, Val. A *state* is a member of Σ_V. A *stream* is a member of $\mathit{Stream}_V \mathrel{\widehat{=}} \mathit{Time} \to \Sigma_V$, which formalises the behaviour of a system over all time. A *predicate* over a type X is given by $\mathcal{P}X \mathrel{\widehat{=}} X \to \mathbb{B}$. Hence, a *state predicate* is a member of $\mathcal{P}\Sigma_V$ and a *stream predicate* is a member of $\mathcal{P}\mathit{Stream}_V$. An *interval predicate* is a member of $\mathit{IntvPred}_V \mathrel{\widehat{=}} \mathit{Interval} \to \mathcal{P}\mathit{Stream}_V$, which defines a property of the given stream with respect to the given interval. Because streams define behaviours over all time, it is possible to use interval predicates to formalise behaviours *outside* a given interval (c.f., operators prev and next below).

One must often reason about properties that hold over two portions of an interval [**?**, **?**], over all subintervals of an interval [**?**] and about properties that hold before and after an interval (c.f. neighbourhood logic [**?**]). Thus, we define the following operators, where p, p_1, p_2 are interval predicates and Δ is an interval.

$$(p_1 \,;\, p_2).\Delta.s \mathrel{\widehat{=}} \exists \Delta' \in \mathit{prefix}.\Delta \bullet (\Delta \neq \Delta') \wedge p_1.\Delta'.s \wedge p_2.(\Delta \backslash \Delta').s$$
$$(\Box p).\Delta.s \mathrel{\widehat{=}} \forall \Delta' \in \mathit{Interval} \bullet \Delta' \subseteq \Delta \Rightarrow p.\Delta'.s$$
$$(\mathsf{prev}.p).\Delta.s \mathrel{\widehat{=}} \exists \Delta' \in \mathit{Interval} \bullet (\Delta' \propto \Delta) \wedge p.\Delta'.s$$
$$(\mathsf{next}.p).\Delta.s \mathrel{\widehat{=}} \exists \Delta' \in \mathit{Interval} \bullet (\Delta \propto \Delta') \wedge p.\Delta'.s$$

The *chop* operator ';' allows the given interval to be split into two so that p_1 holds for the first part and p_2 holds for the second. Note that unlike Duration Calculus [**?**], our chop operator does not require that the two chopped intervals are closed and hence overlap by one point.[1] Interval predicate $\Box p$ holds iff p holds for every subinterval of the given interval, $\mathsf{prev}.p$ holds iff p holds for some interval that immediately precedes the given interval, and $\mathsf{next}.p$ holds iff p holds for some interval that immediately follows the given interval. We define the *weak chop* operator ':' that allows either p_1 to hold for the given interval, or the interval to be split into two so that $(p_1 \,;\, p_2)$ holds in the interval.

$$(p_1 : p_2).\Delta.s \mathrel{\widehat{=}} p_1.\Delta.s \vee (p_1 \,;\, p_2).\Delta.s$$

The Boolean operators may be lifted pointwise to state, stream, and interval predicates. Thus, for example the disjunction $(p_1 \vee p_2).\Delta.s$ is equivalent to $(p_1.\Delta.s \vee p_2.\Delta.s)$ for any interval predicates p_1 and p_2. This allows one to abbreviate a definition like that for weak chop above to

$$p_1 : p_2 \mathrel{\widehat{=}} p_1 \vee (p_1 \,;\, p_2)$$

in which the interval Δ and stream s are implicit and the occurrence of "$\vee$" is lifted disjunction. We make use of such lifted operators from here on to allow definitions and equations to be expressed more succinctly.

When reasoning about properties of programs, one would like to state that whenever a property p_1 holds over any interval Δ and stream s, a property p_2 also holds over Δ and s. Hence, we define universal implication over intervals and streams as follows.

$$p_1.\Delta \Rrightarrow p_2.\Delta \mathrel{\widehat{=}} \forall s \in \mathit{Stream} \bullet p_1.\Delta.s \Rightarrow p_2.\Delta.s$$
$$p_1 \Rrightarrow p_2 \mathrel{\widehat{=}} \forall \Delta \in \mathit{Interval} \bullet p_1.\Delta \Rrightarrow p_2.\Delta$$

We say $p_2 \Lleftarrow p_1$ holds iff $p_1 \Rrightarrow p_2$ holds, and say $p_1 \equiv p_2$ holds iff both $p_1 \Rrightarrow p_2$ and $p_2 \Rrightarrow p_1$ hold. These relations are lower in precedence than any other operators.

The following lemma relates chop to a weak chop using the lengths of the intervals under consideration.

Lemma 3.1. For interval predicates p and q and reals L_1 and L_2 such that $L_1 \geq 0$ and $L_2 > 0$, we have

$$(\ell \geq L_1 + L_2) \wedge ((\ell \leq L_1 \wedge p) : q) \Rrightarrow (\ell \leq L_1 \wedge p) \,;\, (\ell \geq L_2 \wedge q) \tag{1}$$

[1] Frameworks that include infinite length intervals often define chop so that $(p_1 \,;\, p_2).\Delta.s$ also holds if $\mathsf{lub}.\Delta = \infty$ and $p_1.\Delta.s$ holds [**?**, **?**, **?**, **?**], which essentially allows p_1 to model infinite behaviour. However, this generalised definition of chop is not necessary here because we restrict our attention to finite length intervals.

Fig. 4. Splits and prefix-closed

The lemma below allows simplification of interval predicates that use $next$ and chop operators.

Lemma 3.2. For any interval predicate p,

$$true\ ;\ (\mathsf{next}.p) \Rrightarrow \mathsf{next}.p \tag{2}$$

We define some properties on state and interval predicates that are required to enable compositional proofs. Fig. **??** illustrates properties splits and prefix-closed properties of an interval predicate p that are useful for proof decomposition. Provided that p splits, $p.\Delta'$ holds for any subinterval $\Delta' \subseteq \Delta$ whenever $p.\Delta$ holds, and provided that p is prefix-closed, $p.\Delta'$ holds for any prefix Δ' of Δ whenever $p.\Delta$ holds.

Definition 3.1. Suppose p is an interval predicate. We say

1. p *splits* in Δ iff $p.\Delta \Rrightarrow (\Box p).\Delta$, and
2. p is *prefix closed* in Δ iff $p.\Delta \Rrightarrow \forall \Delta' \in \mathit{prefix}.\Delta \bullet p.\Delta'$.

We say p *splits* and p is *prefix closed* iff p splits in Δ and p is prefix closed in Δ, respectively, for every interval Δ.

Note that if p splits in Δ then p is prefix-closed in Δ. The lemma below presents distributivity properties for interval predicates over the chop operator.

Lemma 3.3. For interval predicates p, p_1, p_2, q and r such that p, q and $\neg q$ are prefix closed and r splits, each of the following hold.

$$p \wedge (p_1\ ;\ p_2) \Rrightarrow (p \wedge p_1)\ ;\ p_2 \tag{3}$$
$$r \wedge (p_1\ ;\ p_2) \Rrightarrow (r \wedge p_1)\ ;\ (r \wedge p_2) \tag{4}$$
$$\mathsf{next}.q : \neg q \equiv \mathsf{next}.q \tag{5}$$

Proof. The proofs of (**??**) and (**??**) are straightforward and are omitted here. For (**??**), by definition of weak chop ':', the proof of $\mathsf{next}.q \Rrightarrow \mathsf{next}.q : \neg q$ is trivial. To prove the other direction, we have the following calculation for some arbitrarily chosen interval Δ.

$$\begin{array}{ll}
 & (\mathsf{next}.q\ ;\ \neg q).\Delta \\
\Rrightarrow & \quad \text{definition of ';'} \\
 & \exists \Delta_1, \Delta_2 \bullet (\Delta_1 \propto \Delta_2) \wedge \mathsf{next}.q.\Delta_1 \wedge \neg q.\Delta_2 \\
\equiv & \quad \text{definition of next} \\
 & \exists \Delta_1, \Delta_2, \Delta_3 \bullet (\Delta_1 \propto \Delta_2) \wedge (\Delta_1 \propto \Delta_3) \wedge q.\Delta_3 \wedge \neg q.\Delta_2 \\
\Rrightarrow & \quad (\Delta_1 \propto \Delta_2) \wedge (\Delta_1 \propto \Delta_3) \\
 & \exists \Delta_2, \Delta_3 \bullet ((\Delta_2 \in \mathit{prefix}.\Delta_3) \vee (\Delta_3 \in \mathit{prefix}.\Delta_2)) \wedge q.\Delta_3 \wedge \neg q.\Delta_2 \\
\equiv & \quad \text{logic} \\
 & (\exists \Delta_2, \Delta_3 \bullet (\Delta_2 \in \mathit{prefix}.\Delta_3) \wedge q.\Delta_3 \wedge \neg q.\Delta_2) \vee (\exists \Delta_2, \Delta_3 \bullet (\Delta_3 \in \mathit{prefix}.\Delta_2) \wedge q.\Delta_3 \wedge \neg q.\Delta_2) \\
\Rrightarrow & \quad \text{both } q \text{ and } \neg q \text{ are prefix closed} \\
 & \mathit{false}
\end{array}$$

Hence, $\mathsf{next}.q : \neg q \Rrightarrow \mathsf{next}.q$. □

3.3. Always and sometime

For a state predicate c, we say $(\boxdot c).\Delta.s$ if c holds for all states of s within interval Δ. Similarly, we use $(\diamondsuit c).\Delta.s$ to denote that c holds for some state of s within interval Δ, where

$$(\boxdot c).\Delta.s \mathrel{\widehat{=}} \forall t \in \Delta \bullet c.(s.t)$$
$$(\diamondsuit c).\Delta.s \mathrel{\widehat{=}} \exists t \in \Delta \bullet c.(s.t)$$

Example 3.1. We consider the formalisation of the durative actions of the program in Fig. **??**. We assume each object, obj, on the table is associated with a vector, $pos.obj$, that determines the position of the object obj in polar coordinates, i.e., using a *magnitude* and an *angle*. We assume that addition and subtraction of vectors is defined in the standard manner. Furthermore, we assume $rot.robot$ denotes the angle of rotation of the robot, $gdist$ denotes the distance between the fingers of the robot's grippers, and max_gd is a constant for the maximum distance between the grippers. We use κ and ξ to denote the rotational and forward speed of the robot, respectively, and use φ to denote the speed at which the grippers open/close. We obtain the formalisations of the actions below.

$$\mathsf{nil} \mathrel{\widehat{=}} \boxdot(\mathring{rot}.robot = \mathring{pos}.robot = \mathring{gdist} = 0) \tag{6}$$
$$\mathsf{rotate} \mathrel{\widehat{=}} \boxdot(\mathring{rot}.robot = \kappa) \wedge \boxdot(\mathring{pos}.robot = \mathring{gdist} = 0) \tag{7}$$
$$\mathsf{forward} \mathrel{\widehat{=}} \boxdot(\mathring{pos}.robot = (\xi, rot.robot)) \wedge \boxdot(\mathring{rot}.robot = \mathring{gdist} = 0) \tag{8}$$
$$\mathsf{grasp} \mathrel{\widehat{=}} \boxdot(\mathring{gdist} = (\mathsf{if}\ gdist > 0\ \mathsf{then}\ -\varphi\ \mathsf{else}\ 0)) \wedge \boxdot(\mathring{rot}.robot = \mathring{pos}.robot = 0) \tag{9}$$
$$\mathsf{ungrasp} \mathrel{\widehat{=}} \boxdot(\mathring{gdist} = (\mathsf{if}\ gdist < max\_gd\ \mathsf{then}\ \varphi\ \mathsf{else}\ 0)) \wedge \boxdot(\mathring{rot}.robot = \mathring{pos}.robot = 0) \tag{10}$$

We assume that the program is an abstraction of a real system, and hence may safely assume that $rot.robot$, $pos.robot$ and $gdist$ are continuous. However, we assume idealised behaviour and hence, the variables $\mathring{rot}.robot$, $\mathring{pos}.robot$ and $\mathring{gdist}$ that correspond to the rates of change are discrete, but are piecewise continuous.

The nil action resets the rates of change, i.e., the robot and its grippers do not move while the nil action is executing. The rotate action (**??**) ensures that the robot rotates at a constant angular velocity κ without changing its position or the distance between the grippers. The forward action, (**??**), modifies the position of the robot at the velocity with magnitude ξ and angle $rot.robot$. The angle of rotation of the robot, i.e., $rot.robot$ and gripper distance $gdist$ are not modified by forward. The grasp action guarantees that the rate of change of $gdist$ is $-\varphi$ as long as $gdist$ is above 0 and does not change the position or rotation. The ungrasp action is similar. ♣

Lemma 3.4. For a state predicate c, both of the following hold:

$$\boxdot c \equiv (\boxdot c : \boxdot c) \tag{11}$$
$$\ell > 0 \Rrightarrow \boxdot c = (\boxdot c\ ;\ \boxdot c) \tag{12}$$

Both properties above hold because $\boxdot c$ splits and hence is also prefix closed.

3.4. Values at the limits

In this paper, intervals may be open or closed at either end and variables are assumed to be piecewise continuous [**?**]. Hence, defining the value of variable at the ends of an interval needs care. For instance, suppose we are interested in defining the value of variable v at the right end of an interval Δ. If Δ is right closed, the value of v at the right end of Δ is simply its value at the least upper bound of Δ. However, if Δ is right open, because $\mathsf{lub}.\Delta \notin \Delta$, one must take the limit of v approaching $\mathsf{lub}.\Delta$. Because, we only assume piecewise continuity, it is possible for the values of v approaching $\mathsf{lub}.\Delta$ from the left and right to differ (e.g., if there is a point of discontinuity at $\mathsf{lub}.\Delta$). To ensure that the right limit value of v for Δ is sensible, we take the limit of v approaching $\mathsf{lub}.\Delta$ from the left. A similar argument applies to the value of v at the left end of Δ.

We use $\lim_{x \to a^-} f.x$ and $\lim_{x \to a^+} f.x$ to denote the limit of $f.x$ as x tends to a from the left and right, respectively. For any variable v, interval Δ and stream s, we define the following operators:

$$\overleftarrow{v}.\Delta.s \mathrel{\widehat{=}} \begin{cases} (s.(\mathsf{glb}.\Delta)).v & \text{if } \mathsf{glb}.\Delta \in \Delta \\ \lim\limits_{t \to \mathsf{glb}.\Delta^+} (s.t).v & \text{otherwise} \end{cases} \qquad \overrightarrow{v}.\Delta.s \mathrel{\widehat{=}} \begin{cases} (s.(\mathsf{lub}.\Delta)).v & \text{if } \mathsf{lub}.\Delta \in \Delta \\ \lim\limits_{t \to \mathsf{lub}.\Delta^-} (s.t).v & \text{otherwise} \end{cases}$$

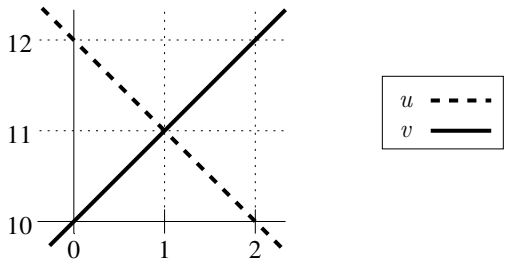

Fig. 5. Variables u and v over interval $[0, 2]$ with $\mathring{u} = -1$ and $\mathring{v} = 1$

where $\overleftarrow{v}$ returns the limit value of v in s at the greatest lower bound of the given interval Δ if Δ is left closed, and the value of v as it approaches the left limit of Δ from the right, if Δ is left open (similarly $\overrightarrow{v}$).

Lemma 3.5. For a continuous variable v and constant k

$(\overrightarrow{v} = k) \equiv \mathsf{next}.(\overleftarrow{v} = k)$

Example 3.2. Lemma **??** does not hold for variables that are only piecewise continuous for intervals starting/ending at a point of discontinuity. Suppose v is a piecewise continuous variable and s is a stream such that $\boxdot(v = 10).[0, 1].s$ and $\boxdot(v = 11).(1, 2]$. Then, both $(\overrightarrow{v} = 10).[0, 1].s$ and $(\overleftarrow{v} = 11).(1, 2].s$ hold but $\mathsf{next}.(\overleftarrow{v} = 10).[0, 1]$ does not hold. ♣

As with variables, we must often refer to the limit values of a state predicate. However, unlike variables, state predicates are boolean-valued, and hence are discrete. For a state predicate c, interval Δ and stream s we define:

$$\overleftarrow{c}.\Delta.s \mathrel{\widehat{=}} \begin{cases} c.(s.(\mathsf{glb}.\Delta)) & \text{if } \mathsf{glb}.\Delta \in \Delta \\ \lim\limits_{t \to \mathsf{glb}.\Delta^+} c.(s.t) & \text{otherwise} \end{cases} \qquad \overrightarrow{c}.\Delta.s \mathrel{\widehat{=}} \begin{cases} c.(s.(\mathsf{lub}.\Delta)) & \text{if } \mathsf{lub}.\Delta \in \Delta \\ \lim\limits_{t \to \mathsf{lub}.\Delta^-} c.(s.t) & \text{otherwise} \end{cases}$$

Note that for a variable v and constant k, $(\overleftarrow{v} = k)$ may not imply $\overleftarrow{v = k}$, and vice versa.

Example 3.3. For example, consider the continuous variable v in Fig. **??**. In Fig. **??**, $(s.0).v = 10$ and $(\boxdot\mathring{v}).[0, 2] = 1$ hold. Hence, both $(\overleftarrow{v} = 11).(1, 2].s$ and $(\overleftarrow{v \neq 11}).(1, 2].s$ hold. Clearly, the value of v at time 1 is 11. Hence, the left limit of the variable v within $(1, 2]$ is 11 because the value of the v will be arbitrarily close to 11 as we approach greatest lower bound 1 from the right. However, the left limit of the predicate $v = 11$ is *false* because $v = 11$ is evaluated in states within the interval $(1, 2]$, and $v \neq 11$ for each of these states. However, for the closed interval $[1, 2]$, both $(\overleftarrow{v} = 11).[1, 2].s$ and $(\overleftarrow{v = 11}).[1, 2].s$ hold because the value of v in state $s.1$ is 11. ♣

It is often useful to specify that a state predicate holds at the start of an interval that immediately follows the given interval. Thus for a state predicate c, we define

$\odot c \mathrel{\widehat{=}} \mathsf{next}.\overleftarrow{c}$

It is also useful to specify that a variable v is stable, i.e., that the value of v does not change from its value at the end of some previous interval. Hence, we define interval predicate $\mathsf{stable}.v$ as follows:

$\mathsf{stable}.v \mathrel{\widehat{=}} \exists k \in \mathit{Val} \bullet \mathsf{prev}.(\overrightarrow{v} = k) \wedge \boxdot(v = k)$

That is, $\mathsf{stable}.v$ holds iff for some k, the value of v at the right end of the previous interval is k and $v = k$ holds throughout the current interval. Our definition of stable is necessary because adjoining intervals are disjoint. Hence, to link values of variables between successive intervals, we must consider their values at the end of the previous interval. By Lemma **??**, for a continuous variable v, we have

$\mathsf{stable}.v \equiv (\exists k \bullet \boxdot(v = k)) \equiv (\exists k \bullet \boxdot(v = k) \wedge \mathsf{next}.(\overleftarrow{v} = k))$

As highlighted by Example **??**, there is a fundamental difference between the limit of a variable v being equal to a value k and the limit of the predicate $v = k$. Hence, even though $\mathsf{stable}.v$ for a continuous variable guarantees $\mathsf{next}.(\overleftarrow{v} = k)$ the value of v may never be equal to k in any next interval. We can only deduce that $v = k$ holds at

the start of the next interval if the stronger property $\odot(v = k)$ holds. Hence, we say a variable v is *right stable* iff $\mathsf{right\_stable}.v$ holds, where

$$\mathsf{right\_stable}.v \mathrel{\widehat{=}} \exists k\colon Val \bullet \overrightarrow{v = k} \wedge \odot(v = k)$$

Lemma 3.6. For a state predicate c and interval predicates p_1 and p_2, we have

$$(p_1 \wedge \odot c)\,;\ p_2 \equiv p_1\,;\ (\overleftarrow{c} \wedge p_2) \tag{13}$$

3.5. Zeno-like behaviour

Because we have a dense notion of time, it is possible for interval predicates to specify Zeno-like behaviour, e.g., a state predicate may switch between true to false an infinite number of times within a finite interval. A specification that allows Zeno-like behaviour is not problematic because a real system will not behave in a Zeno-like manner. However, one must take care not to require Zeno-like behaviour, which would mean the system specification is unimplementable.

We assume sequences have natural number indices (starting from 0) and may be infinite.

Definition 3.2 (Partition). The set of all *partitions* of an interval Δ is given by

$$\Pi.\Delta \mathrel{\widehat{=}} \{z \in \mathrm{seq}\,.Interval \mid (\Delta = \bigcup \mathrm{ran}\,.z) \wedge (\forall i \in \mathrm{dom}.z \backslash \{0\} \bullet z.(i-1) \propto z.i)\}$$

The set of *non-Zeno partitions* of Δ is given by

$$\Pi_{NZ}.\Delta \mathrel{\widehat{=}} \{z \in \Pi.\Delta \mid \mathrm{dom}.z \neq \mathbb{N}\}$$

Thus, z is a non-Zeno partition of Δ iff z has a finite number of elements.

Definition 3.3 (Alternates). For a state predicate c, interval Δ, partition $z \in \Pi.\Delta$ and stream s, we define

$$alt.c.z.s \quad \mathrel{\widehat{=}} \quad \forall i \in \mathrm{dom}.z \bullet (\boxdot c \vee \boxdot\neg c).(z.i).s \wedge (i > 0 \Rightarrow ((\boxdot c).(z.(i-1).s) = (\neg\boxdot c).(z.i).s))$$

Thus, $alt.c.z.s$ holds iff z contains a single interval Δ and either $(\boxdot c).\Delta.s$ or $(\boxdot\neg c).\Delta.s$ holds, or c alternates between $\boxdot c$ and $\boxdot\neg c$ holding within the partition z.

Definition 3.4 (Non-Zeno). A state predicate c is *non-Zeno* in stream s within interval Δ, denoted $(NZ.c).\Delta.s$, iff there exists a $z \in \Pi_{NZ}.\Delta$ such that $alt.c.z.s$ holds.

Note that, if such a z exists then it is unique.

4. Formalising teleo-reactive programs

The syntax of teleo-reactive programs is formalised as follows.

Definition 4.1. If V is a set of variables, p is a prefix-closed interval predicate and c is a state predicate, then the abstract syntax of a *teleo-reactive program* is given by TP below.

$$TP ::= \mathbf{out}\ V \bullet SP \qquad SP ::= p \mid \mathrm{seq}\,.GP \qquad GP ::= c \to SP$$

Thus, a teleo-reactive program consists of a set of output variables, V, and a simple program. A simple program may either be a primitive action (given by a prefix-closed interval predicate) or a sequence of guarded simple programs. We follow the convention of using P for a teleo-reactive program, M for a simple program and S for a (possibly empty) sequence of guarded programs. We define functions $in, out \in TP \to \mathbb{P}\ Var$ that return the sets of input and output variables of the given program, respectively. Given that a sequence can be explicitly defined using brackets '$\langle$' and '$\rangle$', and '$\frown$' is the sequence concatenation operator, we define the following functions. We assume $vars.p$ and $vars.c$ denote the sets of free variables of interval predicate p and state predicate c, respectively.

$$in.(\mathbf{out}\ V \bullet \mathsf{M}) \mathrel{\widehat{=}} cin.V.\mathsf{M} \qquad out.(\mathbf{out}\ V \bullet \mathsf{M}) \mathrel{\widehat{=}} V$$

where

$$cin.V.p \mathrel{\widehat{=}} vars.p \backslash V$$
$$cin.V.\langle\rangle \mathrel{\widehat{=}} \{\}$$
$$cin.V.(\langle c \to \mathsf{M}\rangle \frown S) \mathrel{\widehat{=}} vars.c \cup cin.V.\mathsf{M} \cup cin.V.S$$

We assume that inputs and outputs are disjoint by requiring $in.\mathsf{P} \cap out.\mathsf{P} = \{\}$ for any teleo-reactive program P. In particular, this ensures that $vars.c \cap out.T = \{\}$ holds for any guard c within program P.

Definition 4.2. The behaviour of a program with respect to a stream and interval is given by function beh, which is defined recursively as follows, where $T \mathrel{\widehat{=}} \langle c \to \mathsf{M} \rangle \frown S$.

$$beh.p \mathrel{\widehat{=}} p \tag{14}$$

$$beh.\langle\rangle \mathrel{\widehat{=}} true \tag{15}$$

$$beh.T \mathrel{\widehat{=}} NZ.c \Rightarrow ((\boxdot c \wedge beh.\mathsf{M}) : (\overleftarrow{\neg c} \wedge beh.T)) \vee ((\boxdot \neg c \wedge beh.S) : (\overleftarrow{c} \wedge beh.T)) \tag{16}$$

$$beh.(\mathbf{out}\ V \bullet \mathsf{M}) \mathrel{\widehat{=}} beh.\mathsf{M} \tag{17}$$

By (**??**), the behaviour of a primitive action p is given by the interval predicate p itself. The behaviour of an empty sequence of programs, (**??**), is chaotic, i.e., it allows any behaviour. The behaviour of a sequence of guarded programs (**??**) is defined recursively and requires that the state predicate c is non-Zeno in order to avoid chaotic behaviour. Because the guard c involves inputs to the teleo-reactive program, the condition that c is non-Zeno becomes an assumption about the environment in which the program runs. There are two disjuncts corresponding to either c or $\neg c$ holding initially on the interval. If c holds initially, either $\boxdot c \wedge beh.\mathsf{M}$ holds for the whole interval or the interval may be split into an initial interval in which $\boxdot c \wedge beh.\mathsf{M}$ holds, followed by an interval in which $\neg c$ holds initially and $beh.T$ holds (recursively) for the second interval. The other disjunct is similar. Note that each chopped interval must be a maximal interval over which $\boxdot c$ or $\boxdot \neg c$ holds. For the programs that are developed in this paper, we avoid chaotic behaviour by assuming that all guards are non-Zeno and require that the disjunction of all guards in a sequence holds. The simplest way to achieve the latter is by using $true$ as the last guard in any sequence of guarded programs.

Provided that $\langle c \to \mathsf{M} \rangle \frown S$ executes within Δ and that c alternates in a non-Zeno partition z of Δ, one can deduce that for each interval $z.i$ of z, either $(\boxdot c \wedge beh.\mathsf{M}).(z.i)$ or $(\boxdot \neg c \wedge beh.S).(z.i)$ holds. This is formalised by the following lemma.

Lemma 4.1. If $T \mathrel{\widehat{=}} \langle c \to \mathsf{M} \rangle \frown S$ then for any interval Δ and stream s,

$$\forall z \in \Pi_{NZ}.\Delta \bullet alt.c.z.s \wedge beh.T.\Delta.s \Rightarrow \forall i \in \mathrm{dom}.z \bullet (\boxdot c \wedge beh.\mathsf{M}).(z.i).s \vee (\boxdot \neg c \wedge beh.S).(z.i).s\ . \tag{18}$$

Because primitive actions must be prefix closed, and the definition of the behaviour of a sequence of actions preserves prefix closure if its constituent programs are prefix closed, we have the following lemma.

Lemma 4.2. For any program P or simple program M its behaviour is *prefix closed*.

Informally speaking, prefix closure states that if a program P behaves as specified by $beh.\mathsf{P}$ over an interval Δ, then it must execute as specified by $beh.\mathsf{P}$ in any prefix of Δ. It turns out that any sensible program specification is prefix closed, i.e., programs that are not prefix closed tend to specify unimplementable behaviour. However, if $(beh.\mathsf{P}).\Delta$ holds, then it is generally not necessary for $(beh.\mathsf{P}).\Delta'$ to hold for every subinterval $\Delta' \subseteq \Delta$. That is, there are perfectly sensible programs for which $(beh.\mathsf{P}).\Delta$ does not imply $(beh.\mathsf{P}).\Delta'$ where $\Delta' \subseteq \Delta$. For example, suppose P is a program that causes a component to accelerate to some operating speed. Then $(beh.\mathsf{P}).\Delta$ does not necessarily imply $(beh.\mathsf{P}).\Delta'$ for any arbitrary $\Delta' \subseteq \Delta$, however, if $\Delta' \in prefix.\Delta$, then $(beh.\mathsf{P}).\Delta$ does imply $(beh.\mathsf{P}).\Delta'$.

If $\boxdot c$ holds in the interval over which $\langle c \to \mathsf{M} \rangle \frown S$ executes, the program must be behaving as M over the interval. Similarly, if $\boxdot \neg c$ holds, then the program must be behaving as S. This is formalised by the following lemma.

Lemma 4.3 (Program reduction).

$$\boxdot c \Rrightarrow (beh.(\langle c \to \mathsf{M} \rangle \frown S) = beh.\mathsf{M}) \tag{19}$$

$$\boxdot \neg c \Rrightarrow (beh.(\langle c \to \mathsf{M} \rangle \frown S) = beh.S) \tag{20}$$

Proof. The proofs of both cases follow trivially by definition (**??**) of $beh.(\langle c \to \mathsf{M} \rangle \frown S)$. □

5. Rely/guarantee

We assume that teleo-reactive programs execute within a continually evolving environment. For example, the environment of the robot in Fig. **??** may add or remove cans from both the table and the robot's grasp. In particular, the environment may act maliciously, e.g., by always removing cans from the robot's grasp so that the robot is never able

to deposit cans into the depot. Thus, in order to build robust systems that take the environment into account, we use rely/guarantee-style reasoning [**?**]. Here the *rely* condition describes properties of the environment and the *guarantee* condition describes how the program will behave under the assumption that the rely condition holds. The program does not ensure the guarantee condition outside of the rely condition. In this paper, rely and guarantee conditions are interval predicates. Thus, we may reason about safety, progress and real-time properties of the system within a single formalism.

A teleo-reactive program may not depend on its own output, and hence, the rely condition of a program may not refer to the output variables of the program. A guarantee condition may be a relationship between the inputs and outputs. Within program $\mathsf{P} \mathrel{\widehat{=}} \mathbf{out}\ V \bullet \mathsf{M}$, the simple program M and all programs within M execute in the output context V. To ensure that the rely conditions we develop for the programs within M (including M itself) is a valid rely condition of P, we require that V and the set of the (free) variables of r are disjoint.

Definition 5.1. For a teleo-reactive program $\mathbf{out}\ V \bullet \mathsf{M}$, the *output context* of each simple program in M (including M itself) is the set of variables V.

Definition 5.2. For any simple program M with output context V, an interval predicate r is a *rely condition* of M if $vars.r \cap V = \{\}$.

Rely/guarantee reasoning is defined for both teleo-reactive and simple programs. Thus, we use a more generic P for a program, which may be a teleo-reactive or simple program. Our notation is similar to Hoare-triples for axiomatic proofs of sequential programs [**?**].

Definition 5.3. Suppose P is a program with rely condition r and q is an interval predicate representing the guarantee of P. We define:

$$\{r\}\ \mathsf{P}\ \{q\} \mathrel{\widehat{=}} r \wedge beh.\mathsf{P} \Rrightarrow q$$

Hence, $\{r\}\ \mathsf{P}\ \{q\}$ is only well-defined if r does not refer to any output variables of P. By expanding $\Rrightarrow$ twice, we have

$$\{r\}\ \mathsf{P}\ \{q\} = \forall \Delta \in Interval \bullet \forall s \in Stream \bullet (r \wedge beh.\mathsf{P}).\Delta.s \Rightarrow q.\Delta.s$$

i.e., for any interval Δ and any stream s, if the rely condition r holds and the program behaves as specified by $beh.\mathsf{P}$ over Δ in s, then the guarantee holds over Δ in s.

Lemma 5.1 (Weaken rely, strengthen guarantee). If $r \Rrightarrow r'$ and $\{r'\}\ \mathsf{P}\ \{q'\}$ and $q' \Rrightarrow q$, then $\{r\}\ \mathsf{P}\ \{q\}$.

Lemma 5.2 (Rely disjunction, guarantee conjunction). Both of the following hold:

$$(\{r_1\}\ \mathsf{P}\ \{q\}) \wedge (\{r_2\}\ \mathsf{P}\ \{q\}) \Leftrightarrow \{r_1 \vee r_2\}\ \mathsf{P}\ \{q\} \tag{21}$$

$$(\{r\}\ \mathsf{P}\ \{q_1\}) \wedge (\{r\}\ \mathsf{P}\ \{q_2\}) \Leftrightarrow \{r\}\ \mathsf{P}\ \{q_1 \wedge q_2\} \tag{22}$$

6. Progress

For a goal-directed teleo-reactive program, our aim is to show that the program will achieve its goal within a specified time assuming that the environment does not negatively impact on the program. The assumptions about the environment become part of the rely condition. More specifically we aim to prove results of the form

$$\{r \wedge \ell \geq L\}\ T\ \{\overleftarrow{d} \wedge \boxdot h \Rightarrow \diamondsuit g \vee \odot g\} \tag{23}$$

where r is a rely condition, $L \geq 0$ is a time, state predicate g is the goal of the program, state predicate d is some initial condition and state predicate h is a condition that, in the main result, will turn out to be an accumulation of context from the higher-levels of a hierarchically nested teleo-reactive program. Informally, this formula states that, for a long enough interval, either the goal becomes true in the interval or it will become true immediately after the interval. We simplify such formulae by eliminating the need for the $\diamondsuit$ operator using the following equivalence.

$$\begin{aligned}
& \boxdot h \Rightarrow (\diamondsuit g \vee \odot g) \\
\equiv\ & \boxdot h \Rightarrow (\neg\diamondsuit g \Rightarrow \odot g) \\
\equiv\ & \boxdot h \wedge \neg\diamondsuit g \Rightarrow \odot g \\
\equiv\ & \boxdot h \wedge \boxdot\neg g \Rightarrow \odot g \\
\equiv\ & \boxdot(h \wedge \neg g) \Rightarrow \odot g
\end{aligned}$$

We therefore aim to prove results of the form

$$\{r \wedge \ell \geq L\}\ T\ \{\overleftarrow{d}\ \wedge \boxdot(h \wedge \neg g) \Rightarrow \odot g\} \tag{24}$$

The next lemma considers a program of the form $T \mathrel{\widehat{=}} \langle c \rightarrow \mathsf{M}\rangle \frown S$ in the case where c is initially true. If M achieves the goal g while c is maintained and, while executing M, the environment either achieves the goal or maintains c then executing T will achieve the goal.

Lemma 6.1 (Progress). Suppose $T \mathrel{\widehat{=}} \langle c \rightarrow \mathsf{M}\rangle \frown S$ is a simple program with rely condition r such that $r \Rrightarrow NZ.c$ and r is prefix-closed. For state predicates h and g, and time L such that $L \geq 0$, if both of the following hold

$$\{r \wedge (\ell \geq L)\}\ \mathsf{M}\ \{\boxdot(h \wedge c \wedge \neg g) \Rightarrow \odot g\} \tag{25}$$

$$\{r\}\ \mathsf{M}\ \{\boxdot(h \wedge c \wedge \neg g) \Rightarrow \odot((h \wedge c) \vee g)\} \tag{26}$$

then

$$\{r \wedge (\ell \geq L)\}\ T\ \{\overleftarrow{c}\ \wedge \boxdot(h \wedge \neg g) \Rightarrow \odot g\} \tag{27}$$

Proof. We assume $r \wedge (\ell \geq L) \wedge beh.T \wedge \overleftarrow{c}\ \wedge \boxdot(h \wedge \neg g)$ and prove $\odot g$.

$r \wedge (\ell \geq L) \wedge beh.T \wedge \overleftarrow{c} \wedge \boxdot(h \wedge \neg g)$
$\Rightarrow$ definition of beh using $\overleftarrow{c}$ and $r \Rrightarrow NZ.c$
$r \wedge (\ell \geq L) \wedge ((\boxdot c \wedge beh.\mathsf{M}) : (\overleftarrow{\neg c} \wedge beh.T)) \wedge \boxdot(h \wedge \neg g)$
$\equiv$ definition of weak chop
$(r \wedge (\ell \geq L) \wedge \boxdot c \wedge beh.\mathsf{M} \wedge \boxdot(h \wedge \neg g)) \vee$
$(r \wedge (\ell \geq L) \wedge ((\boxdot c \wedge beh.\mathsf{M})\ ;\ (\overleftarrow{\neg c} \wedge beh.T)) \wedge \boxdot(h \wedge \neg g))$
$\Rightarrow$ (**??**) and logic
$\odot g \vee (r \wedge ((\boxdot c \wedge beh.\mathsf{M})\ ;\ (\overleftarrow{\neg c} \wedge beh.T)) \wedge \boxdot(h \wedge \neg g))$
$\Rightarrow$ r is prefix closed and (**??**), $\boxdot(h \wedge \neg g)$ splits and (**??**)
$\odot g \vee ((r \wedge \boxdot(h \wedge c \wedge \neg g) \wedge beh.\mathsf{M})\ ;\ (\boxdot(h \wedge \neg g) \wedge \overleftarrow{\neg c}))$
$\Rightarrow$ (**??**)
$\odot g \vee (\odot((h \wedge c) \vee g)\ ;\ (\boxdot(h \wedge \neg g) \wedge \overleftarrow{\neg c}))$
$\Rightarrow$ (**??**) and logic
$\odot g \vee (true\ ;\ \overleftarrow{c \wedge \neg c})$
$\equiv$ logic
$\odot g$ □

The next lemma states that if a program T, executing on a sufficiently long interval, can achieve c then intervals on which $\neg c$ is maintained when executing T are bounded.

Lemma 6.2 (Achieve). Suppose T is a simple program with rely condition r such that r is prefix-closed. For state predicates d, h and c, and time $L \geqslant 0$, if the following holds

$$\{r \wedge (\ell \geqslant L)\}\ T\ \{\overleftarrow{d}\ \wedge \boxdot(h \wedge \neg c) \Rightarrow \odot c\} \tag{28}$$

then

$$\{r\}\ T\ \{\overleftarrow{d}\ \wedge \boxdot(h \wedge \neg c) \Rightarrow \ell \leqslant L\} \tag{29}$$

Proof. We assume $r \wedge \overleftarrow{d}\ \wedge beh.T \wedge \boxdot(h \wedge \neg c)$ and prove $\ell \leqslant L$

$r \wedge \overleftarrow{d}\ \wedge beh.T \wedge \boxdot(h \wedge \neg c)$
$\Rightarrow$ $(\ell \leqslant L) \vee (\ell > L)$
$(\ell \leqslant L) \vee (r \wedge (\ell > L) \wedge \overleftarrow{d}\ \wedge beh.T \wedge \boxdot(h \wedge \neg c))$
$\Rightarrow$ $\ell > L \Rrightarrow (\ell \geqslant L)\ ;\ true$
$(\ell \leqslant L) \vee (r \wedge ((\ell \geqslant L)\ ;\ true) \wedge beh.T \wedge \overleftarrow{d}\ \wedge \boxdot(h \wedge \neg c))$
$\Rightarrow$ r and $beh.T$ are prefix closed, (**??**) and (**??**) twice
$(\ell \leqslant L) \vee ((r \wedge (\ell \geqslant L) \wedge beh.T \wedge \overleftarrow{d}\ \wedge \boxdot(h \wedge \neg c))\ ;\ \boxdot(h \wedge \neg c))$
$\Rightarrow$ (**??**)

$$
\begin{array}{ll}
& (\ell \leqslant L) \vee (\odot c\,;\ \boxdot(h \wedge \neg c)) \\
\equiv & \quad \textbf{(??)} \\
& (\ell \leqslant L) \vee (\mathit{true}\,;\ (\overleftarrow{c} \wedge \boxdot(h \wedge \neg c))) \\
\equiv & \quad \overleftarrow{c} \wedge \boxdot\neg c \equiv \mathit{false} \\
& (\ell \leqslant L)
\end{array}
$$

□

The following theorem allows progress properties (i.e. progression towards a goal) to be decomposed to the level of actions automatically. Note that formula (**??**) could be simplified by removing the second occurrence of h but is left in this form so that this formula matches the pattern of formula (**??**). Furthermore, $\overleftarrow{true}$ can be added to formula (**??**) so it also matches the pattern of formula (**??**). Thus the theorem can be repeatedly applied to both S and M.

Theorem 6.1 (Progression). Suppose $T \mathrel{\widehat{=}} \langle c \rightarrow \mathsf{M}\rangle \frown S$ is a simple program with rely condition r such that $r \Rrightarrow NZ.c$ and r splits and hence is prefix-closed. For state predicates d, h and g, and times L_1 and L_2 such that $0 \leq L_1$ and $0 < L_2$, then

$$\{r \wedge (\ell \geq L_1 + L_2)\}\ T\ \{\overleftarrow{d} \wedge \boxdot(h \wedge \neg g) \Rightarrow \odot g\} \tag{30}$$

holds provided

$$\{r \wedge (\ell \geq L_1)\}\ S\ \{\overleftarrow{d} \wedge \boxdot(h \wedge \neg((h \wedge c) \vee g)) \Rightarrow \odot((h \wedge c) \vee g)\} \tag{31}$$

$$\{r \wedge (\ell \geq L_2)\}\ \mathsf{M}\ \{\boxdot(h \wedge c \wedge \neg g) \Rightarrow \odot g\} \tag{32}$$

$$\{r\}\ \mathsf{M}\ \{\boxdot(h \wedge c \wedge \neg g) \Rightarrow \odot((h \wedge c) \vee g)\} \tag{33}$$

Proof. We assume $r \wedge (\ell \geq L_1 + L_2) \wedge \mathit{beh}.T \wedge \overleftarrow{d} \wedge \boxdot(h \wedge \neg g)$ and prove $\odot g$ by performing the following case analysis:

- Case $\overleftarrow{c}$. We have the following calculation:

$$
\begin{array}{ll}
& r \wedge (\ell \geq L_1 + L_2) \wedge \mathit{beh}.T \wedge \overleftarrow{d \wedge c} \wedge \boxdot(h \wedge \neg g) \\
\equiv & \quad \text{Lemma **??** with (**??**) and (**??**)} \\
& \odot g
\end{array}
$$

- Case $\overleftarrow{\neg c}$. We have the following calculation:

$$
\begin{array}{ll}
& r \wedge (\ell \geq L_1 + L_2) \wedge \mathit{beh}.T \wedge \overleftarrow{d \wedge \neg c} \wedge \boxdot(h \wedge \neg g) \\
\equiv & \quad \text{definition of } \mathit{beh} \text{ using } \overleftarrow{\neg c} \text{ and } r \Rrightarrow NZ.c \\
& r \wedge (\ell \geq L_1 + L_2) \wedge ((\boxdot\neg c \wedge \mathit{beh}.S) : (\overleftarrow{c} \wedge \mathit{beh}.T)) \wedge \overleftarrow{d \wedge \neg c} \wedge \boxdot(h \wedge \neg g) \\
\Rrightarrow & \quad r \text{ splits, (**??**), (**??**), noting } \boxdot(h \wedge \neg g) \wedge \boxdot\neg c \equiv \boxdot(h \wedge \neg((h \wedge c) \vee g)) \\
& (\ell \geq L_1 + L_2) \wedge (r \wedge \mathit{beh}.S \wedge \overleftarrow{d} \wedge \boxdot(h \wedge \neg((h \wedge c) \vee g))) : (r \wedge \overleftarrow{c} \wedge \mathit{beh}.T \wedge \boxdot(h \wedge \neg g))) \\
\Rrightarrow & \quad \text{Lemma **??** and (**??**)} \\
& \quad \text{instantiating the } T \text{ of Lemma **??** to } S \text{ and the } c \text{ of Lemma **??** to } (h \wedge c) \vee g \\
& (\ell \geq L_1 + L_2) \wedge ((\ell \leqslant L_1) : (r \wedge \overleftarrow{c} \wedge \mathit{beh}.T \wedge \boxdot(h \wedge \neg g)) \\
\Rrightarrow & \quad \text{(**??**) using } L_2 > 0 \\
& ((\ell \leqslant L_1)\,;\ (r \wedge (\ell \geqslant L_2) \wedge \overleftarrow{c} \wedge \mathit{beh}.T \wedge \boxdot(h \wedge \neg g)) \\
\Rrightarrow & \quad \text{Lemma **??** with (**??**) and (**??**)} \\
& (\ell \leqslant L_1)\,;\ \odot g \\
\Rrightarrow & \quad \textbf{(??)} \\
& \odot g
\end{array}
$$

□

The following lemma can be used to expand (**??**).

Lemma 6.3. Suppose $T \mathrel{\widehat{=}} \langle c \rightarrow \mathsf{M}\rangle \frown S$ is a simple program with rely condition r such that $r \Rrightarrow NZ.c$ and r splits and hence is prefix-closed. For state predicates c_1, c_2 and g, then

$$\{r\}\ T\ \{\boxdot(c_1 \wedge \neg c_2 \wedge \neg g) \Rightarrow \odot(c_1 \vee g)\} \tag{34}$$

holds provided

$$\{r\}\ \mathsf{M}\ \{\boxdot(c \wedge c_1 \wedge \neg c_2 \wedge \neg g) \Rightarrow \odot((c_1 \wedge c_2) \vee (c_1 \wedge c) \vee g)\} \tag{35}$$

$$\{r\}\ S\ \{\boxdot(\neg c \wedge c_1 \wedge \neg c_2 \wedge \neg g) \Rightarrow \odot(c_1 \vee g)\} \tag{36}$$

Proof. We assume $r \wedge beh.T \wedge \boxdot(c_1 \wedge \neg c_2 \wedge \neg g)$ and prove $\odot(c_1 \vee g)$ by performing the following case analysis:

- Case $\overleftarrow{c}$. We have the following calculation:

$$
\begin{array}{ll}
 & \overleftarrow{c} \wedge r \wedge beh.T \wedge \boxdot(c_1 \wedge \neg c_2 \wedge \neg g) \\
\equiv & \quad \text{definition of } beh \text{ using } \overleftarrow{c} \text{ and } r \text{ splits} \\
 & (r \wedge beh.\mathsf{M} \wedge \boxdot(c_1 \wedge c \wedge \neg c_2 \wedge \neg g)) : (\overleftarrow{\neg c} \wedge r \wedge beh.T \wedge \boxdot(c_1 \wedge \neg c_2 \wedge \neg g)) \\
\Rrightarrow & \quad \textbf{(??)} \\
 & (\odot((c_1 \wedge c_2) \vee (c_1 \wedge c) \vee g)) : (\overleftarrow{\neg c} \wedge r \wedge beh.T \wedge \boxdot(c_1 \wedge \neg c_2 \wedge \neg g)) \\
\equiv & \quad \textbf{(??)} \\
 & \odot((c_1 \wedge c_2) \vee (c_1 \wedge c) \vee g) \\
\Rrightarrow & \\
 & \odot(c_1 \vee g)
\end{array}
$$

- Case $\overleftarrow{\neg c}$. We have the following calculation:

$$
\begin{array}{ll}
 & \overleftarrow{\neg c} \wedge r \wedge beh.T \wedge \boxdot(c_1 \wedge \neg c_2 \wedge \neg g) \\
\equiv & \quad \text{definition of } beh \text{ using } \overleftarrow{\neg c} \text{ and } r \text{ splits} \\
 & (r \wedge beh.S \wedge \boxdot(c_1 \wedge \neg c \wedge \neg c_2 \wedge \neg g)) : (\overleftarrow{c} \wedge r \wedge beh.T \wedge \boxdot(c_1 \wedge \neg c_2 \wedge \neg g)) \\
\equiv & \quad \text{definition of weak chop} \\
 & (r \wedge beh.S \wedge \boxdot(c_1 \wedge \neg c \wedge \neg c_2 \wedge \neg g)) \vee \\
 & ((r \wedge beh.S \wedge \boxdot(c_1 \wedge \neg c \wedge \neg c_2 \wedge \neg g)) \,;\, (\overleftarrow{c} \wedge r \wedge beh.T \wedge \boxdot(c_1 \wedge \neg c_2 \wedge \neg g))) \\
\Rrightarrow & \quad \text{`;' is monotonic, } \textbf{(??)} \text{ and the case above} \\
 & \odot(c_1 \vee g) \vee (\mathit{true} \,;\, \odot(c_1 \vee g)) \\
\equiv & \quad \textbf{(??)} \text{ and logic} \\
 & \odot(c_1 \vee g)
\end{array}
$$

□

Example 6.1. Now consider the teleo-reactive program $\langle c \to \langle f \to N\rangle \frown U\rangle \frown S$ where N, U and S are simple teleo-reactive programs. We can use Theorem **??** (progression) repeatedly to replace

$$\{r \wedge \ell \geq L_1 + L_2 + L_3\}\ \langle c \to \langle f \to N\rangle \frown U\rangle \frown S\ \{\overleftarrow{d} \wedge \boxdot\neg g \Rightarrow \odot g\} \tag{37}$$

with the following triples where $NU \mathrel{\widehat{=}} \langle f \to N\rangle \frown U$.

$$\{r \wedge \ell \geq L_1\}\ \ S\ \ \{\overleftarrow{d} \wedge \boxdot\neg(c \vee g) \Rightarrow \odot(c \vee g)\} \tag{38}$$
$$\{r \wedge \ell \geq L_2\}\ \ U\ \ \{\boxdot(c \wedge \neg(f \vee g)) \Rightarrow \odot((c \wedge f) \vee g)\} \tag{39}$$
$$\{r \wedge \ell \geq L_3\}\ \ N\ \ \{\boxdot(c \wedge f \wedge \neg g) \Rightarrow \odot g\} \tag{40}$$
$$\{r\}\ NU\ \{\boxdot(c \wedge \neg g) \Rightarrow \odot(c \vee g)\} \tag{41}$$
$$\{r\}\ \ N\ \ \{\boxdot(c \wedge f \wedge \neg g) \Rightarrow \odot((c \wedge f) \vee g)\} \tag{42}$$

By applying Lemma **??** above, (**??**) holds if both of the following hold.

$$\{r\}\ N\ \{\boxdot(c \wedge f \wedge \neg g) \Rightarrow \odot((c \wedge f) \vee g)\} \tag{43}$$
$$\{r\}\ U\ \{\boxdot(c \wedge \neg(f \vee g)) \Rightarrow \odot(c \vee g)\} \tag{44}$$

Notice that (**??**) and (**??**) are equivalent. In fact, it turns out that, if Theorem **??** (progression) is used to completely unfold all subprograms that are not primitive actions, the rely/guarantee triples involving primitive actions that come from (**??**) are the same triples as those obtained from non-primitive actions in (**??**) via repeated unfolding using Lemma **??**.

Another way to achieve this rewrite is by

1. flattening a hierarchical teleo-reactive program and making all guards explicit so that every subprogram of the top-level program is a primitive action, then
2. repeatedly applying Theorem **??** (progression) on this flattened program.

The rely/guarantee triples on actions generated by this process are the same as the fully unfolded triples generated from the original hierarchical teleo-reactive program. Hence, when fully unfolding triples for a hierarchical teleo-reactive program using Theorem **??** (progression), one can safely ignore the triples obtained from (**??**) on non-primitive actions.

```
prog_rule(rg(R ∧ ℓ ≥ L1 + L2, [C ~> M | S], ←D ∧ ⊡(H ∧ ¬G) ⇒ ⊙G),
          [rg(R ∧ ℓ ≥ L1, S, ←D ∧ ⊡(H ∧ ¬((H ∧ C) ∨ G)) ⇒ ⊙(H ∧ C ∨ G),
           rg(R ∧ ℓ ≥ L2, M, ⊡(H ∧ C ∧ ¬G) ⇒ ⊙G)],
          rg(R, M, ⊡((H ∧ C ∧ ¬G) ⇒ ⊙(H ∧ C ∨ G)).
prog_rule(rg(R ∧ ℓ ≥ L1 + L2, [C ~> M | S], ⊡(H ∧ ¬G) ⇒ ⊙G),
          [rg(R ∧ ℓ ≥ L1, S, ⊡(H ∧ ¬((H ∧ C) ∨ G)) ⇒ ⊙(H ∧ C ∨ G),
           rg(R ∧ ℓ ≥ L2, M, ⊡(H ∧ C ∧ ¬G) ⇒ ⊙G)],
          rg(R, M, ⊡((H ∧ C ∧ ¬G) ⇒ ⊙(H ∧ C ∨ G)).

gen_rg_formulae(Program, Rely, Length, D, H, G, Simp_Rules, Simp_MU): −
    expand([rg(Rely ∧ ℓ ≥ Length, Program, ←D ∧ ⊡(H ∧ ¬G) ⇒ ⊙G)], Rules, MU),
    simplify_rgs(Rules, Simp_Rules),
    simplify_rgs(MU, Simp_MU).
```

Fig. 6. Prolog program for generating proof obligations on the actions

7. Mechanisation

Mechanisation clearly is necessary for large-scale proofs because human-based management of proof obligations quickly becomes infeasible. Theorem **??** facilitates recursive decomposition of proof obligations for proving progress and has been tailored to suit to mechanisation.

To develop the mechanisation, we have a choice of encoding our logic into a theorem prover and developing tactics based on our theorems. However, the proof obligations for Theorem **??** may be generated via a straightforward pattern-matching program that may be implemented in Prolog. The pattern-matching program (see Appendix **??**) decomposes the proof to the level of the primitive actions, and thus avoids many of the errors encountered during a manual proof because a user is only required to instantiate the rely conditions at the level of the primitive actions. A pretty-printed version of the Prolog code is given in Fig. **??** and provides a prototype for the development of the full mechanisation. We note that in the Prolog implementation '∧' has a higher precedence than '∨'.

The two prog_rule predicates instantiate Theorem **??**. The first prog_rule matches guarantees expanded using (**??**) (that include a conjunct $\overleftarrow{d}$) and the second matches guarantees expanded using (**??**) (in which conjunct $\overleftarrow{d} = true$). Within the first prog_rule, the first argument corresponds to (**??**), the second argument corresponds to a list containing proof obligations (**??**) and (**??**), and the third argument corresponds to (**??**).

To run the Prolog program, we call gen_rg_formulae, which takes a program Program, rely condition Rely, overall time bound Length, initial condition D, accumulation of high-level guards H and goal G. The expand predicate repeatedly applies prog_rule whereby we obtain a list of rely/guarantee triples on the actions, Rules, and a list of rely/guarantee triples with 'maintain unless' predicates, MU. We simplify the rely/guarantee triples that we obtain using predicate simplify_rgs, which reduces predicates such as $true \wedge p$ to p. We may also assert known properties of the program (not shown), which are taken into account during the simplification, e.g., in the robot example, we have $may\_hold\_can \Rightarrow see\_can$, which may be used to simplify $see\_can \wedge may\_hold\_can$ to just may_hold_can.

8. Can clearing robot

In this section, we apply the theory in Section **??** to the can clearing robot example. We describe the relationships between inputs and outputs and specify the requirements of the example in Section **??**. In Section **??**, we describe the outputs of our mechanisation for the robot example. Then, in Section **??**, we describe how the rely condition may be instantiated so that the generated proof obligations are discharged.

8.1. Specification

Guard specification. We use the following definitions to determine whether or not the robot sees or holds the given object. We assume $rot.robot$ denotes the angle of rotation of the robot, and $pos.robot \approx pos.can$ holds iff the touching

sensor activates, which occurs iff the robot is sufficiently close to can to be touching it. We define:

$$\begin{aligned} sees.obj &\mathrel{\widehat{=}} (pos.obj - pos.robot).angle \bmod 2\pi = rot.robot \bmod 2\pi \\ may\_hold.can &\mathrel{\widehat{=}} sees.can \wedge (pos.robot \approx pos.can) \\ holds.can &\mathrel{\widehat{=}} may\_hold.can \wedge (gdist = can\_width) \end{aligned}$$

where can_width is a constant for the width of the can. Thus, the robot can see obj iff the angle of the vector from the position of the robot to the position of obj is equal to the rotational angle of the robot, the robot may hold can iff their positions are within $\approx$ and it sees can, and the robot holds can iff it may hold can (i.e., sees and touches can) and the distance between the grippers is can_width. To ensure the robot is able to grasp a can, we require that

$$0 < can\_width < max\_gd \tag{45}$$

i.e., the grippers are able to open past can_width. The relationships between the guards in Fig. **??** and the state of the system is given by the following properties, where $pos.depot$ is constant and TC is the set of all cans on the table.

$$\begin{aligned} sensors \mathrel{\widehat{=}} & (depot\_empty = (\forall\, can \in TC \bullet pos.can \neq pos.depot)) \wedge \\ & (see\_can = \exists can \in TC \bullet sees.can) \wedge \\ & (see\_depot = sees.depot) \wedge \\ & (may\_hold\_can = \exists can \in TC \bullet may\_hold.can) \wedge \\ & (holding = \exists can \in TC \bullet holds.can) \wedge \\ & (at\_depot = see\_depot \wedge (pos.robot \approx pos.depot)) \wedge \\ & (open = (gdist = max\_gd)) \end{aligned}$$

Thus, for example, see_can holds iff there is a can on the table such that the robot can see the can.

Requirement specification. We define

$$can\_exists \mathrel{\widehat{=}} TC \neq \{\}$$

which states that there is at least one can on the table. Our progress requirement is that the following must hold for some (yet to be derived) interval predicates R, and time L. Condition R is necessary because it is typically not possible to prove (**??**) in an arbitrary environment. For example, the environment may remove all the cans from the table, which falsifies can_exists without establishing progress.

$$\{R \wedge (\ell \geq L)\}\ \mathsf{robot}\ \{\overleftarrow{can\_exists} \Rightarrow \odot(\neg depot\_empty \wedge (\neg at\_depot \vee open))\} \tag{46}$$

We implicitly assume idealised inputs by assuming:

$$R \Rrightarrow \boxdot sensors \tag{47}$$

8.2. Proof of progress

The repeated expansions of sequences of guarded actions that are necessary to prove proof (**??**) are automated using the Prolog program in Fig. **??**. To this end, the robot program is encoded as a Prolog list within the predicate robot_program in Fig. **??** where all nested simple programs have been expanded to the level of the actions.

We run the Prolog program to generate the proof obligations by executing

```
|?- robot_program(Program),
   gen_rg_formulae
       (Program, R, L, can_exists, true, ¬depot_empty ∧ (¬at_depot ∨ open), Rules, MU).
```

The first output Rules of the program is provided in Fig. **??**, which lists the progress conditions required of the individual actions. We also obtain:

$$\mathsf{L} = \mathsf{A} + \mathsf{B} + \mathsf{C} + \mathsf{D} + \mathsf{E} + \mathsf{F} + \mathsf{G}$$

which represents a bound on the time taken to achieve the overall goal. That is, the maximum time taken to deliver the can is the sum of the maximum times taken to achieve the subgoals. The second output MU of the program is given in Fig. **??**, which lists the proof obligations for a state predicate to be maintained. Note that there is no condition corresponding to rotate within collect because the guard to be maintained is just $true$.

$$
\begin{array}{l}
|? - \mathsf{robot\_program}([\neg \mathit{depot\_empty} \wedge (\neg \mathit{at\_depot} \vee \mathit{open}) \sim\!> \mathsf{nil},\\
\qquad\qquad\qquad \neg \mathit{depot\_empty} \sim\!> \mathsf{ungrasp},\\
\qquad\qquad\qquad \mathit{holding} \sim\!> [\mathit{see\_depot} \sim\!> \mathsf{forward},\\
\qquad\qquad\qquad\qquad\qquad \mathit{true} \sim\!> \mathsf{rotate}],\\
\qquad\qquad\qquad \mathit{true} \sim\!> [\mathit{see\_can} \sim\!> [\mathit{may\_hold\_can} \sim\!> \mathsf{grasp},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{open} \sim\!> \mathsf{forward},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{true} \sim\!> \mathsf{ungrasp}],\\
\qquad\qquad\qquad\qquad\quad \mathit{true} \sim\!> \mathsf{rotate}]]).
\end{array}
$$

Fig. 7. Program for mechanisation

Condition (**??**) corresponds to rotate within collect; (**??**), (**??**) and (**??**) correspond to ungrasp, forward and grasp within fetch; (**??**) and (**??**) correspond to rotate and forward within go_depot; and (**??**) to ungrasp within robot.

$$
\begin{array}{lll}
\mathsf{Rules} = & & \\
[\mathsf{rg}(\mathsf{R} \wedge \ell \geq \mathsf{A}, & \mathsf{rotate}, & \overleftarrow{\mathit{can\_exists}} \wedge \boxdot(\neg \mathit{see\_can} \wedge \mathit{depot\_empty}) \Rightarrow \\
& & \quad \odot(\mathit{see\_can} \vee \neg \mathit{depot\_empty})), \qquad (48)\\
\mathsf{rg}(\mathsf{R} \wedge \ell \geq \mathsf{B}, & \mathsf{ungrasp}, & \boxdot(\neg \mathit{open} \wedge \neg \mathit{may\_hold\_can} \wedge \mathit{see\_can} \wedge \mathit{depot\_empty}) \Rightarrow \\
& & \quad \odot((\mathit{open} \wedge \mathit{see\_can}) \vee \mathit{may\_hold\_can} \vee \neg \mathit{depot\_empty})), \qquad (49)\\
\mathsf{rg}(\mathsf{R} \wedge \ell \geq \mathsf{C}, & \mathsf{forward}, & \boxdot(\mathit{see\_can} \wedge \mathit{open} \wedge \neg \mathit{may\_hold\_can} \wedge \mathit{depot\_empty}) \Rightarrow \\
& & \quad \odot(\mathit{may\_hold\_can} \vee \neg \mathit{depot\_empty})), \qquad (50)\\
\mathsf{rg}(\mathsf{R} \wedge \ell \geq \mathsf{D}, & \mathsf{grasp}, & \boxdot(\mathit{may\_hold\_can} \wedge \neg \mathit{holding} \wedge \mathit{depot\_empty}) \Rightarrow \\
& & \quad \odot(\mathit{holding} \vee \neg \mathit{depot\_empty})), \qquad (51)\\
\mathsf{rg}(\mathsf{R} \wedge \ell \geq \mathsf{E}, & \mathsf{rotate}, & \boxdot(\mathit{holding} \wedge \neg \mathit{see\_depot} \wedge \mathit{depot\_empty}) \Rightarrow \\
& & \quad \odot((\mathit{holding} \wedge \mathit{see\_depot}) \vee \neg \mathit{depot\_empty})), \qquad (52)\\
\mathsf{rg}(\mathsf{R} \wedge \ell \geq \mathsf{F}, & \mathsf{forward}, & \boxdot(\mathit{holding} \wedge \mathit{see\_depot} \wedge \mathit{depot\_empty}) \Rightarrow \\
& & \quad \odot\neg \mathit{depot\_empty}), \qquad (53)\\
\mathsf{rg}(\mathsf{R} \wedge \ell \geq \mathsf{G}, & \mathsf{ungrasp}, & \boxdot(\mathit{at\_depot} \wedge \neg \mathit{open} \wedge \neg \mathit{depot\_empty}) \Rightarrow \\
& & \quad \odot((\neg \mathit{at\_depot} \vee \mathit{open}) \wedge \neg \mathit{depot\_empty}))] \qquad (54)
\end{array}
$$

Fig. 8. Proof obligations for each action to make progress

Note that each automatically generated proof obligation in Fig. **??** and Fig. **??** is at the level of the primitive actions, and hence requires no further expansion.

8.3. Deriving the rely condition

In this section, we discharge the proof obligations in Figs. **??** and **??** by instantiating a sufficiently strong rely condition.

Proof obligations in Fig. ??.

*Proof of (**??**).* We prove the triple below, which implies (**??**).

$$\{\mathsf{R} \wedge (\ell \geq \mathsf{A})\}\ \mathsf{rotate}\ \{\overleftarrow{\mathit{can\_exists}} \wedge \boxdot\neg \mathit{see\_can} \Rightarrow \odot \mathit{see\_can}\} \qquad (61)$$

That is, if the robot is rotating for at least A time units and R holds over the interval, then the behaviour of rotate guarantees that the robot sees a can within or immediately after the given interval, provided that there is a can on the table at the beginning of the interval. Clearly this property cannot hold if the environment continually moves the cans out of view of the robot. Hence, using the fact that $\mathit{pos.can}$ is right stable, we consider instantiating the rely condition R so that the following holds:

$$\mathsf{R} \Rrightarrow \Box(\exists \mathit{can} \in \mathit{TC} \bullet \mathsf{stable}.(\mathit{pos.can}) \wedge \mathsf{right\_stable}.(\mathit{pos.can}))$$

$$
\begin{array}{lll}
\mathsf{MU} = & & \\
\quad [\mathsf{rg}(\mathsf{R},\ \mathsf{ungrasp}, & \boxdot(\neg open \wedge \neg may\_hold\_can \wedge see\_can \wedge depot\_empty) \Rightarrow & (55)\\
& \quad \odot(see\_can \vee \neg depot\_empty)), & \\
\quad \mathsf{rg}(\mathsf{R},\ \mathsf{forward}, & \boxdot(see\_can \wedge open \wedge \neg may\_hold\_can \wedge depot\_empty) \Rightarrow & (56)\\
& \quad \odot((see\_can \wedge open) \vee may\_hold\_can \vee \neg depot\_empty)), & \\
\quad \mathsf{rg}(\mathsf{R},\ \mathsf{grasp}, & \boxdot(may\_hold\_can \wedge \neg holding \wedge depot\_empty) \Rightarrow & (57)\\
& \quad \odot(may\_hold\_can \vee \neg depot\_empty)), & \\
\quad \mathsf{rg}(\mathsf{R},\ \mathsf{rotate}, & \boxdot(holding \wedge \neg see\_depot \wedge depot\_empty) \Rightarrow & (58)\\
& \quad \odot(holding \vee \neg depot\_empty)), & \\
\quad \mathsf{rg}(\mathsf{R},\ \mathsf{forward}, & \boxdot(holding \wedge see\_depot \wedge depot\_empty) \Rightarrow & (59)\\
& \quad \odot((holding \wedge see\_depot) \vee \neg depot\_empty)), & \\
\quad \mathsf{rg}(\mathsf{R},\ \mathsf{ungrasp}, & \boxdot(at\_depot \wedge \neg open \wedge \neg depot\_empty) \Rightarrow & (60)\\
& \quad \odot(\neg depot\_empty))] &
\end{array}
$$

Fig. 9. Proof obligations for each action to maintain its guard

i.e., in all intervals there is a can on the table whose position is stable and right stable. However, such a rely condition is too strong because there may not be any cans on the table to begin with. Hence, we weaken the requirement on the rely condition to:

$$\mathsf{R} \Rrightarrow \Box(\overleftarrow{can\_exists} \Rightarrow \exists can \in TC \bullet \mathsf{stable}.(pos.can) \wedge \mathsf{right\_stable}.(pos.can))$$

This property is still too strong because it disallows the robot from moving the can if there is only one can on the table and the robot is holding that can. Thus, we use $\boxdot\neg see\_can$ in the antecedent of the guarantee of (**??**) to further weaken the rely condition, i.e., we obtain:

$$\mathsf{R} \Rrightarrow \Box(\overleftarrow{can\_exists} \wedge \boxdot\neg see\_can \Rightarrow \exists can \in TC \bullet \mathsf{stable}.(pos.can) \wedge \mathsf{right\_stable}.(pos.can)) \quad (62)$$

Thus, for any values of the robot position and rotation, if there is a can on the table at the start of the interval and the robot cannot see a can throughout the interval, there must be at least one can whose position is stable in the interval. We now complete the proof of (**??**) as follows:

$$
\begin{array}{ll}
& (\mathbf{??})\\
\Leftarrow & \quad \text{use } (\mathbf{??})\\
& \{\mathsf{R} \wedge (\ell \geq \mathsf{A})\}\ \mathsf{rotate}\ \left\{(\exists can \in TC \bullet \mathsf{stable}.(pos.can) \wedge \mathsf{right\_stable}.(pos.can)) \wedge \boxdot\neg see\_can \Rightarrow \odot see\_can\right\}\\
\Leftarrow & \quad \text{logic, behaviour of } \mathsf{rotate}\ (\mathbf{??}), \text{ set } \mathsf{A} \geq \tfrac{2\pi}{\kappa} \text{ and } (\mathbf{??})\\
& true
\end{array}
$$

□

The proofs of the remaining obligations of Fig. **??** all follow a shared pattern: for each proof we assume the corresponding proof obligation of Fig. **??** and then use the properties of the action to complete the proof. We now give a detailed proof of (**??**).

*Proof of (**??**).* We assume

$$\mathsf{R} \wedge (\ell \geq \mathsf{B}) \wedge beh.\mathsf{ungrasp} \wedge \boxdot(\neg open \wedge \neg may\_hold\_can \wedge see\_can \wedge depot\_empty)$$

and show $\odot((open \wedge see\_can) \vee may\_hold\_can \vee \neg depot\_empty)$. This is proved as follows:

$$
\begin{array}{ll}
& \mathsf{R} \wedge (\ell \geq \mathsf{B}) \wedge beh.\mathsf{ungrasp} \wedge \boxdot(\neg open \wedge \neg may\_hold\_can \wedge see\_can \wedge depot\_empty)\\
\Rrightarrow & \quad (\mathbf{??})\\
& \mathsf{R} \wedge (\ell \geq \mathsf{B}) \wedge beh.\mathsf{ungrasp} \wedge \boxdot(\neg open \wedge \neg may\_hold\_can \wedge see\_can \wedge depot\_empty)\\
& \wedge \odot(see\_can \vee \neg depot\_empty)\\
\Rrightarrow & \quad \text{logic, behaviour of } \mathsf{ungrasp}\ (\mathbf{??}), \text{ set } \mathsf{B} \geq \tfrac{gdist}{\varphi} \text{ and } (\mathbf{??})\\
& \odot((open \wedge see\_can) \vee \neg depot\_empty)
\end{array}
$$

□

*Proof of (**??**).* We assume

$$\mathsf{R} \Rrightarrow \Box(\forall can \in TC \bullet \boxdot(sees.can \wedge \neg holds.can) \Rightarrow \mathsf{stable}.(pos.can) \wedge \mathsf{right\_stable}.(pos.can)) \tag{63}$$

Thus, if the the robot sees can but does not hold can, (i.e., the grippers are open), then the position of can (that the robot sees) must be stable. We further assume (**??**) and note that, by the behaviour of forward (**??**) and for $\mathsf{C} \geq \frac{max\_dist}{\xi}$ (where max_dist is the maximum distance between any two points on the table), we have

$$\mathsf{R} \wedge (\ell \geq \mathsf{C}) \wedge beh.\mathsf{forward} \wedge \boxdot(see\_can \wedge open) \Rrightarrow \odot may\_hold\_can$$

as required. □

*Proof of (**??**).* We assume (**??**) and (**??**) and note that, for $\mathsf{D} \geq \frac{gdist}{\varphi}$, we have

$$beh.\mathsf{grasp} \wedge \boxdot may\_hold\_can \Rrightarrow \odot holding$$

as required. □

*Proof of (**??**).* We first assume

$$\mathsf{R} \Rrightarrow \Box(\mathsf{stable}.(pos.depot) \wedge \mathsf{right\_stable}.(pos.depot)) \tag{64}$$

We then assume (**??**) and note that, for $\mathsf{E} \geq \frac{2\pi}{\kappa}$, as required, by (**??**) we have:

$$beh.\mathsf{rotate} \Rrightarrow \odot see\_depot$$

□

*Proofs of (**??**) and (**??**).* These are similar to the proofs of (**??**) and (**??**) noting that both of the following hold.

$$\begin{array}{rcl} holding \wedge at\_depot & \Rrightarrow & \neg depot\_empty \\ open \wedge \neg depot\_empty & \Rrightarrow & (\neg at\_depot \vee open) \wedge \neg depot\_empty \end{array}$$

□

Proof obligations in Fig. ??. The formula (**??**) follows from (**??**) as ungrasp does not change the position or direction of the robot and all the cans that are seen do not move and hence $\odot see\_can$. The proof of (**??**) also follows from (**??**) as forward does not change the direction of the robot and hence $\odot see\_can$. The proof of (**??**) is similar as $may\_hold\_can \Rightarrow see\_can$. We now assume

$$\mathsf{R} \Rrightarrow \Box(\boxdot(holding \vee (at\_depot \wedge \neg open \wedge may\_hold\_can)) \Rightarrow \odot may\_hold\_can) \tag{65}$$

The formulae (**??**), (**??**) and (**??**) follow directly.

8.4. Final rely condition

Collecting our assumptions (i.e. (**??**), (**??**), (**??**), (**??**), (**??**)), we instantiate R to

$$\begin{array}{l} \boxdot sensors \wedge \\ \Box(\overleftarrow{can\_exists} \wedge \boxdot\neg see\_can \Rightarrow \exists can \in TC \bullet \mathsf{stable}.(pos.can) \wedge \mathsf{right\_stable}.(pos.can)) \wedge \\ \Box(\forall can \in TC \bullet \boxdot(sees.can \wedge \neg holds.can) \Rightarrow \mathsf{stable}.(pos.can) \wedge \mathsf{right\_stable}.(pos.can)) \wedge \\ \Box(\mathsf{stable}.(pos.depot) \wedge \mathsf{right\_stable}.(pos.depot)) \wedge \\ \Box(\boxdot(holding \vee (at\_depot \wedge \neg open \wedge may\_hold\_can)) \Rightarrow \odot may\_hold\_can) \end{array}$$

Note that this R splits as required because $\boxdot c$ splits for any state predicate c, $\Box p$ splits for any interval predicate p and $p_1 \wedge p_2$ splits if both p_1 and p_2 split. The final rely condition implies each of the following properties.

- The guards are idealised (condition (**??**)). This represents the assumption that sensor values are accurate and there are no delays in turning them on/off.
- If there is a can on the table and the robot does not see any can throughout the given interval, then at least one of the cans must not change its position throughout the interval (condition (**??**)). This ensures that the robot will eventually see a can if it rotates in position.
- If the robot sees a can, say can, and is not holding can throughout the given interval, then the position of can does not change within the interval (condition (**??**)). This ensures that moving forward (towards can) eventually causes robot to touch can. Furthermore, if the robot is already touching and sees can then closing the grippers will cause the robot to hold can.

- The position of the depot does not change in any interval (condition (**??**)). This ensures that rotating in position causes the robot to see the depot. Furthermore, if the robot sees the depot, the moving forward causes the robot to reach the depot.
- If the robot is holding a can, then it continues to hold (or be able to hold) a can and if there is a can at the depot and the robot is at the depot with it's grippers are not fully open then the can must remain in the depot (condition (**??**)).

9. Conclusion

In this paper we have developed a temporal logic on real-time intervals that extends Duration Calculus [**?**], which itself is an extension of Interval Temporal Logic [**?**]. We used our logic to develop a formalisation of the semantics of teleo-reactive programs. Correctness of a teleo-reactive program is judged by considering its behaviour with respect to the environment it operates within, and hence, we present rely/guarantee style specification rules. We have also provided a number of theorems for proving progress in goal-directed agents.

Teleo-reactive programs differ from state-based real-time formalisms such as continuous action systems [**?**, **?**], TLA$^+$ [**?**] and hybrid automata [**?**], where each action is considered instantaneous and causes a discrete state change in the system. Instead, teleo-reactive programs use durative actions that describe a behaviour over an interval of time. Furthermore, teleo-reactive programs have a hierarchical structure, in which the higher-level guards are guaranteed to hold throughout the execution of the component programs. We have assumed that actions are idealised (see (**??**) - (**??**)) and assumed that velocities change instantaneously. However, more sophisticated behaviour that for example incorporates delays and acceleration can be specified [**?**]. Furthermore, like Duration Calculus [**?**], properties that can be specified using straightforward mathematical analysis, which provides one with an interface to standard control theory.

By incorporating durative actions and hierarchical structuring together, teleo-reactive programs can be less complicated than their action systems equivalent [**?**]. Furthermore, each guard in a sequence of guarded programs assumes the negation of all previous guards as an implicit conjunct, which facilitates the development of goal-directed agents [**?**]. Although teleo-reactive programs are simple and compact, the behaviours that the programs specify are complex.

Proving properties of teleo-reactive programs is complicated by the fact that both the program and the environment may affect the guards, and hence, the behaviour of the program. For teleo-reactive programs that implement goal-directed agents, as well as showing that the program moves towards its goal, we must also show that once a program earlier in the sequence is enabled, the guard of the earlier program may not be disabled unless the goal is reached. We have presented a method that clearly separates the two concerns and generates proof obligations for both. Our method is able to generate the requirements on the actions and defer instantiation of the rely condition to a later point. Furthermore, generation of proof obligations on the actions is automated. Instantiation of an appropriate rely condition is simplified when considering properties at the level of the actions.

There are several existing formalisms for reasoning about hybrid systems [**?**, **?**, **?**, **?**, **?**], several of which allow a high degree of automation. As we have already seen in Section **??**, a teleo-reactive program can be represented by these existing frameworks. However, goal-directed reasoning in these frameworks is potentially difficult. The translation of teleo-reactive programs and our progression theorems to an already existing framework in order to make use of the available tool support is an avenue of further research.

We note that our robot example presents an idealised scenario where several physical constraints are simplified. For instance, we assume that acceleration to and from the operating speed is instantaneous (e.g., the robot stops scanning when the can is directly in front of the robot), forward causes the robot to move in a straight line, etc. We could have made our example more complicated by removing these idealised assumptions. However, the purpose of this verification is to demonstrate applicability of our logic for the verification of goal-based agents. We have developed a semantics for teleo-reactive programs [**?**] where the idealised timing assumptions are approximated using time bands [**?**, **?**] and a sampling logic [**?**, **?**]. This has been extended to a rely/guarantee theory that combines interval-based reasoning, sampling and time bands to reason about teleo-reactive programs over multiple time granularities [**?**], however this work does not cover reasoning about progression. Our early experiments indicate that progression in the context of sampling and time bands is a non-trivial task due to the inaccuracies between the observed and actual states. We aim to develop progression theories in such contexts as part of future work.

Acknowledgements. We would like to thank Keith Clark and Kirsten Winter for helpful discussions on early versions of this paper, and our anonymous reviewers for their comments. This research is supported by Australian Research Council (ARC) Discovery Grant DP0987452, The University of Queensland's New Staff Start-up Research Fund, and EPSRC Grant EP/J003727/1.

A. Prolog code for decomposing rely/guarantee triples

```
%
% Prolog program to support the paper
% Reasoning about Goal-Directed Real-Time Teleo-Reactive Programs
%
% Authors: Brijesh Dongol, Ian J. Hayes and Peter J. Robinson
%

?-op(480, fy, ~).
?-op(520, xfy, and).
?-op(530, xfy, or).
?-op(540, xfy, =>).
?-op(525, xfy, mu).
?-op(600, xfy, ~>).

%% Theorem 6.1
prog_rule(rg(R and len(L1+L2), [C ~> M|S],
             ll(DD) and box(H and ~G) => circle(G)),
          [rg(R and len(L1), S,
              ll(DD) and box(H and ~(H and C or G)) => circle(H and C or G)),
           rg(R and len(L2), M, box((H and C) and ~G) => circle(G))],
          rg(R, M, box((H and C) and ~G) => circle(H and C or G))).
prog_rule(rg(R and len(L1+L2), [C ~> M|S], box(H and ~G) => circle(G)),
          [rg(R and len(L1), S,
              box(H and ~(H and C or G)) => circle(H and C or G)),
           rg(R and len(L2), M, box((H and C) and ~G) => circle(G))],
          rg(R, M, box((H and C) and ~G) => circle(H and C or G))).


%% Generate all the formulae from repeated use of the above formulae
%% Simp_Rules and Simp_MU are simplified formulae generated from Theorem 6.1
gen_rg_formulae(Program, Rely, Length, DD, H, G, Simp_Rules, Simp_MU) :-
    expand([rg(Rely and len(Length), Program,
                ll(DD) and box(H and ~G) => circle(G))], Rules, MU),
    simplify_rgs(Rules, Simp_Rules),
    simplify_rgs(MU, Simp_MU).

%% expand(RG, Expand_RG, MU)
%% expands out RG into Expand_RG and MU using Theorem 6.1 repeatedly

expand([], [], []).
% singleton guarded programs
expand([Form|Forms], Expand, MU) :-
    Form = rg(R and len(D), [true ~> A],
              ll(DD) and box(P and ~Q) => circle(Q)),!,
    expand([rg(R and len(D), A,  ll(DD) and box(P and ~Q) => circle(Q))|Forms],
           Expand, MU1),
    MU = [rg(R, A, box(P and ~Q) => circle(P or Q))|MU1].
expand([Form|Forms], Expand, MU) :-
    Form = rg(R and len(D), [true ~> A], box(P and ~Q) => circle(Q)),!,
    expand([rg(R and len(D), A, box(P and ~Q) => circle(Q))|Forms],
           Expand, MU1),
    MU = [rg(R, A, box(P and ~Q) => circle(P or Q))|MU1].
% sequence with at least 2 elemets - use Theorem 6.1
expand([Form|Forms], Expand, MU) :-
    prog_rule(Form, Forms1,MU1),!,
    MU = [MU1|MU2],
    append(Forms1, Forms, AllForms),
    expand(AllForms, Expand, MU2).
expand([Form|Forms], [Form|Expand], MU) :-
    expand(Forms, Expand, MU).
```

```
% Simplify the formulae in a list of RG formulae
simplify_rgs([], []).
simplify_rgs([rg(R,M, box(G) => circle(Q))|Rest], Out) :-
    !,
    simplify(G, SG),
    simplify(Q, SQ),
    simplify_rgs(Rest, SRest),
    (
      (SG = false ; SQ = true)
    ->
      Out = SRest
    ;
      Out = [rg(R,M,box(SG) => circle(SQ))|SRest]
    ).
simplify_rgs([rg(R,M, X and box(G) => circle(Q))|Rest], Out) :-
    !,
    simplify(G, SG),
    simplify(Q, SQ),
    simplify_rgs(Rest, SRest),
    (
      (SG = false ; SQ = true)
    ->
      Out = SRest
    ;
      Out = [rg(R,M,X and box(SG) => circle(SQ))|SRest]
    ).


% Simplify propositional formula involving and, or, => and ~

simplify(Form, SimpForm) :-
    easy_simplify(Form, SForm),
    simplify_formula(SForm, [], SimpForm).

%% Push negations in and simplify away true and false
easy_simplify(~(A and B), SForm) :-
    !,
    easy_simplify(~A, AS),
    easy_simplify(~B, BS),
    easy_simplify_step(AS or BS, SForm).
easy_simplify(~(A or B), SForm) :-
    !,
    easy_simplify(~A, AS),
    easy_simplify(~B, BS),
    easy_simplify_step(AS and BS, SForm).
easy_simplify(~(~A), SForm) :-
    !,
    easy_simplify(A, SForm).
easy_simplify(A and B, SForm) :-
    !,
    easy_simplify(A, AS),
    easy_simplify(B, BS),
    easy_simplify_step(AS and BS, SForm).
easy_simplify(A or B, SForm) :-
    !,
    easy_simplify(A, AS),
    easy_simplify(B, BS),
    easy_simplify_step(AS or BS, SForm).
easy_simplify(~A, SForm) :-
    !,
    easy_simplify(A, AS),
    easy_simplify_step(~AS, SForm).

easy_simplify(A, A).

easy_simplify_step(A and true, A) :- !.
easy_simplify_step(true and A, A) :- !.
```

```
easy_simplify_step(_A or true, true) :- !.
easy_simplify_step(true or _A, true) :- !.
easy_simplify_step(_A and false, false) :- !.
easy_simplify_step(false and _A, false) :- !.
easy_simplify_step(A or false, A) :- !.
easy_simplify_step(false or A, A) :- !.
easy_simplify_step(~(false), true) :- !.
easy_simplify_step(~(true), false) :- !.
easy_simplify_step(A, A).

%% Simplify Form given a collection of Hypotheses
simplify_formula(Form, Hyps, SimpForm) :-
    simplify_formula1(Form, Hyps, SimpForm1, Progress),
    ( var(Progress) ->
        SimpForm = SimpForm1
    ;
        simplify_formula(SimpForm1, Hyps, SimpForm)
    ).

%% Same as simplify_formula but Progress is instantialted to true of some
%% simplification happened

%% A occurs in the hyps
simplify_formula1(A, Hyps, SimpForm, Progress) :-
    member(A, Hyps), !,
    SimpForm = true,
    Progress = true.
%% ~A occurs in the hyps
simplify_formula1(A, Hyps, SimpForm, Progress) :-
    member(~A, Hyps), !,
    SimpForm = false,
    Progress = true.
%% Simplify A assuming B then simplify B assuming the simplification of A
simplify_formula1(A and B, Hyps, SimpForm, Progress) :-
    !,
    simplify_formula1(A, [B|Hyps], SimpA, Progress),
    simplify_formula1(B, [SimpA|Hyps], SimpB, Progress),
    easy_simplify_step(SimpA and SimpB, SimpForm).
%% Simplify A assuming ~B then simplify B assuming the negation of
%% the simplification of A
simplify_formula1(A or B, Hyps, SimpForm, Progress) :-
    !,
    easy_simplify(~B, NB),
    simplify_formula1(A, [NB|Hyps], SimpA, Progress),
    easy_simplify(~SimpA, NA),
    simplify_formula1(B, [NA|Hyps], SimpB, Progress),
    easy_simplify_step(SimpA, SA),
    easy_simplify_step(SimpB, SB),
    easy_simplify_step(SA or SB, SimpForm).
%% Otherwise simplify the hyps and the simplify A with the simplified hyps
simplify_formula1(A, Hyps, SimpForm, Progress) :-
    simp_hyps(Hyps, SimpHyps),
    simp_with_hyps(A, SimpHyps, SimpForm, Progress).

strip_negs(~ ~ A, A) :- !.
strip_negs(A, A).

%% Simplify the hyps
simp_hyps(Hyps, SimpHyps) :-
    expand_conjuncts(Hyps, SimpHyps1),
    add_deductions(SimpHyps1, SimpHyps2),
    find_contrad(SimpHyps2, SimpHyps).
```

```
%% Add all the possible deductions of hyps to the hyps
add_deductions([], []).
add_deductions([A|H], Hyps) :-
    findall(B, deduction(A, B), AD), AD ~= [], !,
    append(AD, H, HypsD),
    add_deductions(HypsD, Hyps1),
    Hyps = [A|Hyps1].
add_deductions([A|H], [A|Hyps]) :-
    add_deductions(H, Hyps).

%% Simplify the hyps to [false] if a contradiction within the hyps exist
find_contrad(H, SH) :-
    member(A, H),
    member(~A, H), !,
    SH = [false].
find_contrad(H, H).

%% expand out conjuncts
expand_conjuncts([], []).
expand_conjuncts([(A and B)|Hyps], SimpHyps) :-
    !, expand_conjuncts([A,B|Hyps], SimpHyps).
expand_conjuncts([A|Hyps], [A|SimpHyps]) :-
    expand_conjuncts(Hyps, SimpHyps).

%% Simplify a formula using the hyps

%% Contradiction - simplify to false
simp_with_hyps(_, [false], false, true) :- !.
%% A is a hyp
simp_with_hyps(A, Hyps, SimpForm, Progress) :-
    member(A, Hyps), !,
    SimpForm = true,
    Progress = true.
%% ~A is a hyp
simp_with_hyps(A, Hyps, SimpForm, Progress) :-
    member(~A, Hyps), !,
    SimpForm = false,
    Progress = true.
simp_with_hyps(~A, Hyps, SimpForm, Progress) :-
    member(A, Hyps), !,
    SimpForm = false,
    Progress = true.
%% Further simplify the hyps and try again
simp_with_hyps(A, Hyps, SimpForm, Progress) :-
    simp_the_hyps(Hyps, [], SHyps, P), nonvar(P), !,
    simp_with_hyps(A, SHyps, SimpForm, Progress).
%% No simplification possible
simp_with_hyps(A, _Hyps, A, _Progress).

%% Simplift the hyps
simp_the_hyps([], Done, Done, _).
simp_the_hyps([false|_Hyps], _Done, [false], true) :- !.
simp_the_hyps([~true|_Hyps], _Done, [false], true) :- !.
simp_the_hyps([true|Hyps], Done, SHyps, P) :-
    !, simp_the_hyps(Hyps, Done, SHyps, P).
simp_the_hyps([~false|Hyps], Done, SHyps, P) :-
    !, simp_the_hyps(Hyps, Done, SHyps, P).
%% If H can be simplified with hyps in Hyps or Done then simplify
simp_the_hyps([H|Hyps], Done, SHyps, P) :-
    can_simplify(H, Hyps, Done), !,
    P = true,
    append(Hyps, Done, All),
    simplify_formula1(H, All, SimpH, _Progress),
    simp_the_hyps(Hyps, [SimpH|Done], SHyps, P).
simp_the_hyps([H|Hyps], Done, SHyps, P) :-
    simp_the_hyps(Hyps, [H|Done], SHyps, P).
```

```
%% can_simplify(A, H1, H2) is true iff A contains a subformula (or a negation
%% of a subformula) that matches a hyp from H1 or H2
can_simplify(A, H1, _H2) :-
    member(A, H1).
can_simplify(A, _H1, H2) :-
    member(A, H2).
can_simplify(~A, H1, _H2) :-
    member(A, H1).
can_simplify(~A, _H1, H2) :-
    member(A, H2).
can_simplify(A, H1, _H2) :-
    member(~A, H1).
can_simplify(A, _H1, H2) :-
    member(~A, H2).
can_simplify(A and _B, H1, H2) :-
    can_simplify(A, H1, H2).
can_simplify(_A and B, H1, H2) :-
    can_simplify(B, H1, H2).
can_simplify(A or _B, H1, H2) :-
    can_simplify(A, H1, H2).
can_simplify(_A or B, H1, H2) :-
    can_simplify(B, H1, H2).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% Can robot specific info
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% deductions: deduction(A, B) means we can deduce B if A is true

deduction(may_hold_can, see_can).
deduction(~see_can, ~ may_hold_can).
deduction(holding, see_can).
deduction(~ see_can, ~holding).
deduction(holding, may_hold_can).
deduction(~may_hold_can, ~holding).
deduction(at_depot, see_depot).
deduction(~see_depot, ~at_depot).

% The can robot program as in the paper
robot_program([~depot_empty and (~at_depot or open) ~> nil,
               ~depot_empty ~> ungrasp,
               holding ~> [see_depot ~> forward,
                           true ~> rotate
                          ],
               true ~> [see_can ~>
                            [may_hold_can ~> grasp,
                             open ~> forward,
                             true ~> ungrasp
                            ],
                        true ~> rotate
                       ]
              ]).

% Flat version of robot program
flat_robot_program([~depot_empty and (~at_depot or open) ~> nil,
                    ~depot_empty ~> ungrasp,
                    holding and see_depot ~> forward,
                    holding ~> rotate,
                    see_can and may_hold_can ~> grasp,
                    see_can and open ~> forward,
                    see_can ~> ungrasp,
                    true ~> rotate
              ]).
```

```
/*
  Generate RG formulae from can robot

  robot_program(Program),
  gen_rg_formulae(Program, R, L, can_exists, true, ~depot_empty and (~at_depot or open), Rules, MU).

  Generate RG formulae from can robot (flat version)

  flat_robot_program(Program),
  gen_rg_formulae(Program, R, L, can_exists, true, ~depot_empty and (~at_depot or open), Rules, MU).
*/

%% For completeness - lemma 6.3
mu_expand(rg(R, [(C ~> M)|S], box((C1 and ~C2) and ~G) => circle(C1 or G)),
          [rg(R, M, box(((C1 and C) and ~C2) and ~G) =>
                       circle(((C1 and C2) or (C1 and C)) or G)),
           rg(R, S, box((C1 and ~(C or C2)) and ~G) => circle(C1 or G))]).
```