THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Reasoning About Real-Time Teleo-Reactive Programs

Brijesh Dongol
Ian J. Hayes
Peter J. Robinson

February 2010

Technical Report SSE-2010-01

Division of Systems and Software Engineering Research
School of Information Technology and Electrical Engineering
The University of Queensland
QLD, 4072, Australia

http://www.itee.uq.edu.au/~sse

Reasoning About Real-Time Teleo-Reactive Programs

Brijesh Dongol, Ian J. Hayes, and Peter J. Robinson

{brijesh,ianh,pjr}@itee.uq.edu.au
School of Information Technology and Electrical Engineering,
The University of Queensland

Abstract. The teleo-reactive programming model is a high-level approach to implementing real-time control programs that react dynamically to changes in their environment. Teleo-reactive programs are particularly useful for implementing controllers in autonomous agents. In this paper we present formal techniques for reasoning about robust teleo-reactive programs. We develop a temporal logic over continuous intervals, which we use to formalise the semantics of teleo-reactive programs. To facilitate compositional reasoning about a program and its environment, we use rely/guarantee style specifications. We also present several theorems for simplifying proofs of teleo-reactive programs that control goal-directed agents.

1 Introduction

Software is increasingly being used to implement controllers for safety-critical applications in real-time environments [14, 11, 5]. For such systems, failures can have a high cost, and hence it is important to ensure dependability of the underlying software throughout the lifetime of the system. As the applications become more sophisticated, the programming languages and the logics that we use must accordingly become more sophisticated.

The teleo-reactive programming language is a high-level language that has been shown to be useful for implementing autonomous agents that react robustly to constantly changing environments [15, 6]. Teleo-reactive programs present several attractive benefits for developing real-time software. Actions are durative in nature, i.e., describe a behaviour over an interval of time, as opposed to formalisms such as Z [17], action systems [1], TLA [10], etc., where each action causes a discrete state change in the system. To reason about continuous properties, these formalisms must be extended so that variables have type $Time \rightarrow Value$ (e.g., as used in continuous action systems [2, 13], TLA$^+$ [9]), where $Time \mathrel{\widehat{=}} \mathbb{R}^{\geq 0}$ denotes the set of all times. Teleo-reactive programs have a hierarchical structure, which means details of each action can be developed at a later stage. Furthermore, several teleo-reactive programs may be composed in parallel, which allows programs to be developed in a modular manner.

We consider a teleo-reactive controller for the robot depicted in Fig. 1, which is tasked with clearing cans from the table by moving them to the depot. To achieve this task, the robot is able to to rotate clockwise/anti-clockwise (to scan the environment for cans or the depot), move in the forward direction (towards a can or the depot), and grasp/ungrasp its fingers (to pickup and drop cans). The robot is equipped with sensors

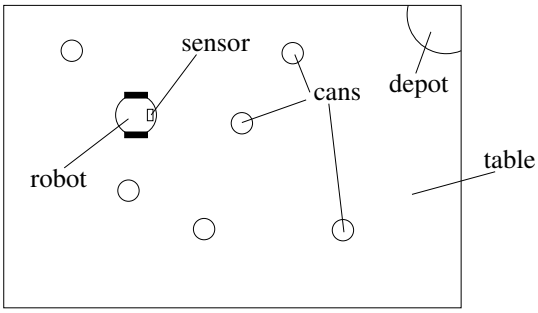

Fig. 1. Top down view of can clearing robot example

$holding$, see_can, and $touching$ that determine whether or not the robot is holding, seeing, and touching a can, respectively. If see_can holds, the robot is pointing at a can, and hence, if the robot moves forward while see_can holds, provided that the environment does not move the can, the robot will eventually touch the can. Using its current location and rotation, the robot is able to derive booleans at_depot and see_depot which hold if the robot is at the depot and can see the depot, respectively.

We assume that the environment does not directly affect movement of the robot, however, it 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 help or hinder the robot from achieving its task.

To reason about teleo-reactive programs, we present a form of temporal logic, which we call *durative temporal logic*. The logic allows reasoning about properties over the duration of time intervals by combining the duration calculus [18] with linear temporal logic [4, 12]. We also use durative temporal logic to formalise the semantics of teleo-reactive programs. To facilitate compositional reasoning, we consider rely/guarantee style specification rules [8], 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 condition holds. Our framework allows us to state rely/guarantee properties that hold over all intervals (to prove safety properties) and intervals of a certain length (to prove progress properties). We may also state properties about the lengths of the intervals (to prove real-time properties). We provide a number of higher-level proof rules for simplifying proofs of progress in goal-directed agents.

In Section 2 we present durative temporal logic as well as a formal syntax and semantics of teleo-reactive programs. In Section 3, we present a compositional theory for proving properties of teleo-reactive programs and in Section 4, we prove properties of a controller for the can clearing robot (Fig. 1).

2 Formalising teleo-reactive programs

In this section we present a theory that facilitates reasoning about teleo-reactive programs. In Section 2.1 we define the durative temporal logic, and in Sections 2.2 and 2.3 we provide the syntax and semantics of teleo-reactive programs, respectively.

2.1 A temporal logic for intervals

We now present a logic for reasoning about the behaviour of teleo-reactive programs. Because the actions of a teleo-reactive program are durative, linear temporal logic [4, 12], which is defined for discrete traces of states is inappropriate. On the other hand, although the duration calculus facilitates reasoning about real-time programs [18], we would like to avoid the complexity introduced by allowing point intervals. Teleo-reactive programs can ignore point intervals because any (primitive) action of a teleo-reactive program is active over an interval with non-zero length. We also introduce new operators that simplify temporal reasoning.

If $b, e \in Time$ and $b < e$, we let $[b, e]$, $(b, e]$, $[b, e)$, and (b, e) denote the *closed*, *right-closed* (or *left-open*), *left-closed* (or *right-open*), and *open* (non-point) interval from b to e, respectively. Note that each interval is finite. The set of all closed, non-point intervals is given by $Intv \mathrel{\widehat{=}} \{[b, e] \mid b, e \in Time \wedge b < e\}$. Hence, $[b', e'] \subseteq [b, e]$ (i.e., $[b', e']$ is a *sub-interval* of $[b, e]$) iff $b \leq b' < e' \leq e$.

A *state space* of a teleo-reactive program is given by $\Sigma \mathrel{\widehat{=}} (Var \cup Env) \rightarrow Value$, where Var and Env are disjoint sets of variable names representing the variables controlled by the program and the environment, respectively. A *state* is a member of Σ, a *state predicate* is a member of $\mathcal{P}\Sigma \mathrel{\widehat{=}} \Sigma \rightarrow \mathbb{B}$, and a *trace* is a member of $Trace \mathrel{\widehat{=}} Time \rightarrow \Sigma$. We use '.' for function application. For a state predicate c, we define

$$\int c \mathrel{\widehat{=}} \lambda [b, e]\colon Intv \bullet \lambda\, tr\colon Trace \bullet \textstyle\int_b^e c.(tr.t)\, dt$$

The *length* of an interval is given by $\ell \mathrel{\widehat{=}} \int 1$. Note that $\ell.[b, e] = e - b$.

A *trace predicate* is a member of $\mathcal{P}\,Trace \mathrel{\widehat{=}} Trace \rightarrow \mathbb{B}$, and an *interval predicate* is a member of $IntvPred \mathrel{\widehat{=}} Intv \rightarrow \mathcal{P}\,Trace$. The boolean operators may be lifted pointwise to state, trace, and interval predicates. Thus, for example $(ip_1 \wedge ip_2).\Delta.tr = (ip_1.\Delta.tr \wedge ip_2.\Delta.tr)$ for interval predicates ip_1 and ip_2. We define some further notation for trace predicates tp_1 and tp_2.

$$(tp_1 \Rrightarrow tp_2) \mathrel{\widehat{=}} \forall tr\colon Trace \bullet tp_1.tr \Rightarrow tp_2.tr$$
$$(ip_1 \Rrightarrow ip_2) \mathrel{\widehat{=}} \forall \Delta\colon Intv \bullet ip_1.\Delta \Rrightarrow ip_2.\Delta$$

(similarly, '$\equiv$' and '$\Lleftarrow$').

Given that c is a state predicate, the syntax of basic durative temporal formulae is described by

$$F ::= \lceil c \rceil \mid F_1\colon F_2 \mid \Box F \mid \circledcirc F$$

and their semantics is given in Fig. 2. We present the semantics in terms of interval predicates and assume that unary operators bind more tightly than binary operators.

The interpretation of $\lceil c \rceil$ is that c holds almost everywhere at the *start* of the given interval. The *chop* operator ':' allows F_1 to hold for the given interval, or we may split the interval into two so that F_1 holds for the first part, and F_2 holds for the second. Note that ':' is weaker than the chop operator defined in the duration calculus [18] in that it allows F_2 not to become true in the given interval. $\Box F$ holds if F holds for every sub-interval of the given interval and $\circledcirc F$ holds if F holds for the interval that immediately follows the given interval.

$$
\begin{aligned}
\lceil c \rceil &\mathrel{\widehat{=}} \lambda[b,e]\colon \mathit{Intv} \bullet \exists m \in (b,e] \bullet (\ell = \textstyle\int c).[b,m] \\
F_1 \colon F_2 &\mathrel{\widehat{=}} \lambda[b,e]\colon \mathit{Intv} \bullet F_1.[b,e] \vee \exists m \in (b,e) \bullet F_1.[b,m] \wedge F_2.[m,e] \\
\Box F &\mathrel{\widehat{=}} \lambda[b,e]\colon \mathit{Intv} \bullet \forall \delta\colon \mathit{Intv} \bullet \delta \subseteq [b,e] \Rightarrow F.\delta \\
\obar F &\mathrel{\widehat{=}} \lambda[b,e]\colon \mathit{Intv} \bullet \exists f > e \bullet F.[e,f]
\end{aligned}
$$

Fig. 2. Durative temporal logic

In order to reduce the notational complexity, we interpret application of a state predicate, say c, to an interval to be equivalent to $\lceil c \rceil$. That is, we remove '$\lceil$' and '$\rceil$' from $\lceil c \rceil$ when the interpretation of c is clear from context. We may view this as lifting a state predicate to an interval predicate.

We also define the following shorthand for reasoning about teleo-reactive programs.

$$
\begin{aligned}
\Diamond F &\mathrel{\widehat{=}} \neg \Box \neg F & \qquad F_1 \ \mathbf{un}\ F_2 &\mathrel{\widehat{=}} F_2 \vee (\Box F_1 \colon F_2) \\
\nabla F &\mathrel{\widehat{=}} \Diamond F \vee \obar F & \qquad F_1 \ \mathbf{wu}\ F_2 &\mathrel{\widehat{=}} F_1 \Rightarrow (F_1 \ \mathbf{un}\ F_2)
\end{aligned}
$$

$\Diamond F$ states that F *eventually* holds in some sub-interval of the given interval and ∇F states that F holds *sometime* within or immediately after the given interval. $F_1 \ \mathbf{un}\ F_2$ states that either F_2 holds, or F_1 continues to hold *unless* F_2 holds. Note that if F_2 never holds, i.e., $\Box \neg F_2$, then $F_1 \ \mathbf{un}\ F_2 \equiv \Box F_1$. Finally, $F_1 \ \mathbf{wu}\ F_2$ is the *weak unless* operator that only requires $F_1 \ \mathbf{un}\ F_2$ to hold if F_1 holds.

2.2 Abstract syntax

We use $\mathrm{seq}\,.T$ to denote a finite sequence with elements of type T. A sequence can be explicitly defined using brackets, '$\langle$' and '$\rangle$', and '$\frown$' is the sequence concatenation operator.

Definition 1. *If $f \subseteq \mathit{Var}$ is a set of variables; r and g are interval predicates; and c is a state predicate, the abstract syntax of a teleo-reactive program is given by*

$$
\begin{aligned}
PA &::= f \colon [\![r, g]\!] \\
GA &::= c \rightarrow A \\
A &::= PA \mid \mathrm{seq}\,.GA
\end{aligned}
$$

A primitive action is defined via a rely/guarantee specification that consists of a *frame*, f, a *rely* condition, r, and a *guarantee* condition, g. Each primitive action, $pa \mathrel{\widehat{=}} f \colon [\![r, g]\!]$, directly controls the variables in f, assumes r holds over any interval in which pa is executing, and ensures that g holds provided r holds. A guarded action $c \rightarrow a$ executes as action a over any interval in which the guard c is true almost everywhere. The interpretation of a sequence of guarded actions is that the earlier actions are given priority over actions that appear later in the sequence. For example, in a sequence $\langle c_1 \rightarrow a_1, c_2 \rightarrow a_2 \rangle$, if the guard c_1 ever becomes true, then a_2 stops and a_1 begins executing. Hence, the guard of a_2 is effectively $\neg c_1 \wedge c_2$. If neither c_1 nor c_2 holds, then

$$\mathsf{robot} \mathrel{\widehat{=}} \boxed{\begin{array}{r} \mathit{holding} \rightarrow \mathsf{deliver}, \\ \mathit{true} \rightarrow \mathsf{collect} \end{array}}$$

$$\mathsf{go\_depot} \mathrel{\widehat{=}} \boxed{\begin{array}{r} \mathit{see\_depot} \rightarrow \mathsf{forward}, \\ \mathit{true} \rightarrow \mathsf{rotate} \end{array}}$$

$$\mathsf{deliver} \mathrel{\widehat{=}} \boxed{\begin{array}{r} \mathit{at\_depot} \rightarrow \mathsf{ungrasp}, \\ \mathit{true} \rightarrow \mathsf{go\_depot} \end{array}}$$

$$\mathsf{fetch} \mathrel{\widehat{=}} \boxed{\begin{array}{r} \begin{pmatrix} \mathit{touching} \wedge \\ \mathit{gdist} > \mathit{can\_width} \end{pmatrix} \rightarrow \mathsf{grasp}, \\ \mathit{gdist} > \mathit{can\_width} \rightarrow \mathsf{forward}, \\ \mathit{true} \rightarrow \mathsf{ungrasp} \end{array}}$$

$$\mathsf{collect} \mathrel{\widehat{=}} \boxed{\begin{array}{r} \mathit{see\_can} \rightarrow \mathsf{fetch}, \\ \mathit{true} \rightarrow \mathsf{rotate} \end{array}}$$

Fig. 3. Control program for a can clearing robot

neither a_1 nor a_2 is executed, and the behaviour is defined to be chaotic [7]. During an execution of a teleo-reactive program, the guards of the program are continually being evaluated, and the highest-priority enabled action is executed.

Let us consider the teleo-reactive program in Fig. 3, which implements a controller for the robot in Fig. 1. We assume that guards $\mathit{holding}$, $\mathit{see\_can}$, $\mathit{at\_depot}$, $\mathit{see\_depot}$, and $\mathit{touching}$ are boolean variables whose values are equivalent to the (sensed or derived) values of the corresponding sensors. We assume that $\mathit{can\_width}$ is the width of the can, $\mathit{max\_gd}$ (where $\mathit{can\_width} < \mathit{max\_gd}$) is the maximum distance between the two gripping fingers of the robot, and gdist (where $0 \leq \mathit{gdist} \leq \mathit{max\_gd}$) is the distance between the two gripping fingers. The primitive actions of the robot are rotate, forward, grasp, and ungrasp, which control the basic movements of the robot.

Durative actions deliver and collect in robot cause the robot to deliver and collect cans, respectively. Action deliver executes over any interval in which $\Box \mathit{holding}$ holds, i.e., over any interval in which the robot continues to hold a can, the robot must be attempting to deliver the can to the depot. Furthermore, deliver is expanded into primitive actions ungrasp and go_depot. In the context of the robot program, the guard of ungrasp is effectively $\mathit{holding} \wedge \mathit{at\_depot}$, and similarly the guard of go_depot is effectively $\mathit{holding} \wedge \neg \mathit{at\_depot}$.

The structure of the program in Fig. 3 is typical of teleo-reactive programs that implement controllers for goal-directed agents. Here, completion of the first action deliver represents accomplishment of the goal, while completion of the second action collect represents achievement of sub-goals that cause the agent to make progress towards enabling (and hence executing) the first action deliver. Notice that actions, deliver, collect, go_depot, and fetch are also structured in a goal-directed manner. For example, completion of the ungrasp action represents completion of deliver (because $\mathit{holding}$ becomes false), while completion of go_depot enables ungrasp. Teleo-reactive programs allow actions to be reused, e.g., rotate occurs in robot (to scan for cans) and in go_depot (to scan for the depot).

Suppose $f \subseteq Var$, r and g are interval predicates, and $\mathsf{M} \mathrel{\widehat{=}} \langle c \rightarrow a \rangle \frown S$ is a teleo-reactive program.

$$beh.(f\colon [\![r, g]\!]) \mathrel{\widehat{=}} r \Rightarrow g \wedge st.(Var \setminus f) \quad (1)$$

$$beh.\langle\rangle \mathrel{\widehat{=}} true \quad (2)$$

$$beh.\mathsf{M} \mathrel{\widehat{=}} ((\Box c \wedge beh.a)\colon (\neg c \wedge beh.\mathsf{M})) \vee ((\Box \neg c \wedge beh.S)\colon (c \wedge beh.\mathsf{M})) \quad (3)$$

Fig. 4. beh function

2.3 Semantics

Recalling the syntax of a teleo-reactive program from Definition 1, the meaning of a teleo-reactive action A over interval Δ is described by the behaviour function:

$$beh\colon A \rightarrow IntvPred$$

as defined in Fig. 4. For a set of variables V, we define an interval predicate, $st.V$, that states that all the variables in V are *stable* over the interval. Formally, we have:

$$st.V \mathrel{\widehat{=}} \forall v\colon V \bullet \exists k\colon Value \bullet \Box(v = k)$$

The behaviour of a primitive action, (1), states that if the rely condition holds, then the guarantee must hold and each variable in Var not in the frame of the primitive action is stable. Note that a primitive action may modify variables from Env. The behaviour of an empty sequence of actions, (2), is chaotic, i.e., any behaviour is allowed. The behaviour of a sequence of guarded actions, (3), is defined recursively. There are two disjuncts corresponding to either c or $\neg c$ holding initially on the interval. If c holds initially, either $\Box c \wedge beh.a$ holds for the whole interval or the interval may be split into an initial interval in which $\Box c \wedge beh.a$ holds, followed by an interval in which $\neg c$ holds initially and $beh.\mathsf{M}$ holds (recursively) for the second interval. The other disjunct is similar. Note that each chopped interval must be a maximal interval over which either $\Box c$ or $\Box \neg c$ holds. The semantics of $beh.\mathsf{M}$ does not rule out Zeno-like behaviour, i.e., an infinite number of switches between different guarded actions over a finite amount of time. However, because Zeno behaviour is not possible in a real system and because our traces are generated from such systems, we assume Zeno-like behaviour does not occur.

3 Compositional proofs of safety and progress

In this section, we describe how proofs of safety and progress may be carried out in a compositional manner. In Section 3.1, we define rely/guarantee conditions and provide lemmas for reducing the complexity of rely/guarantee triples, and in Section 3.2, we describe proof of progress for goal-directed agents.

In this section, we assume c, p and q are state predicates; r, r', g and g' are interval predicates; F and G are durative temporal formulae; $c \rightarrow a$ is a guarded action; S is a sequence of guarded actions; M is a teleo-reactive program; and $D, D_1, D_2 \in Time$.

3.1 Rely/guarantee

Our teleo-reactive programs execute within a continually changing environment. For example, the environment of the robot in Fig. 3 can add or remove cans from both the table and the robot's grasp. Clearly, the environment may act maliciously, e.g., by removing cans from the robot's grasp so that the robot is never able to deposit cans into the depot.

In order to build robust systems that take the actions of the environment into account, we use rely/guarantee specifications [8]. 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 may be any interval predicate (which includes durative temporal formulae). Thus, we may reason about safety, progress and real-time properties of the system within a single formalism. The definition below is provided by Hayes [7].

Definition 2. $\{r\}\mathsf{M}\{g\} \mathrel{\widehat{=}} r \wedge beh.\mathsf{M} \Rrightarrow g$

Note that we have $\{r\}\ \mathsf{M}\ \{g\} = \forall tr\colon Trace \bullet \forall \Delta\colon Intv \bullet (r \wedge beh.\mathsf{M} \Rightarrow g).\Delta.tr$ by expanding $\Rrightarrow$ in Definition 2. Rely condition r may describe conditions under which the environment performs both helpful and harmful state changes.

The following lemma along with the associated corollary allows us to prove that a rely-guarantee triple holds by weakening the rely condition and strengthening the guarantee.

Lemma 3 (Weaken/strengthen). $\{r\}\ \mathsf{M}\ \{g\}$ *holds provided* $(r' \Rightarrow g') \Rrightarrow (r \Rightarrow g)$ *and* $\{r'\}\ \mathsf{M}\ \{g'\}$ *hold.*

Corollary 4 (Weaken/strengthen). $\{r\}\ \mathsf{M}\ \{g\}$ *holds provided* $r \Rrightarrow r'$, $g' \Rrightarrow g$ *and* $\{r'\}\ \mathsf{M}\ \{g'\}$ *hold.*

The following lemma states that we may prove an existential property in the rely condition by proving the triple holds for each possible value of the variable.

Lemma 5. *If* x *is a variable of non-empty type* T *then* $\{\exists x\colon T \bullet r\}\ \mathsf{M}\ \{g\}$ *holds provided* x *is not free in* $beh.\mathsf{M}$ *and* g*, and* $\forall x\colon T \bullet \{r\}\ \mathsf{M}\ \{g\}$ *holds.*

A primitive action, $f\colon [\![r, g]\!]$, will guarantee both that g holds and that the program variables not in f do not change, provided r holds.

Lemma 6 (Primitive action). $\{r\}\ f\colon [\![r, g]\!]\ \{g \wedge st.(Var \setminus f)\}$

The next lemma allows the complexity of the teleo-reactive program within a rely/guarantee triple to be reduced. Given a teleo-reactive program of the form $\langle c \rightarrow a\rangle \frown S$, if $\Box c$ holds in the interval over which $\langle c \rightarrow a\rangle \frown S$ executes, the program must be behaving as a over the interval. Similarly, if $\Box\neg c$ holds, then the program must be behaving as S.

Lemma 7 (Program reduction). *If* $\mathsf{M} \mathrel{\widehat{=}} \langle c \rightarrow a\rangle \frown S$*, then*

$$\{r \wedge \Box c\}\ \mathsf{M}\ \{g\} = \{r \wedge \Box c\}\ a\ \{g\} \tag{4}$$

$$\{r \wedge \Box\neg c\}\ \mathsf{M}\ \{g\} = \{r \wedge \Box\neg c\}\ S\ \{g\} \tag{5}$$

3.2 Progress in goal-directed agents

For teleo-reactive programs that implement goal-directed agents (e.g., Fig. 3), progress consists of showing that the first action in a program is eventually executed. Successful completion of the first action denotes achievement of the overall goal of the program. The rest of the teleo-reactive program ensures that the top action is eventually executed. We work towards Theorem 12 (progression), which allows progress properties of a program to be decomposed to the level of primitive actions. We first present a number of intermediate lemmas.

One way to prove that guarantee condition ∇q holds is to show that an intermediate condition pp is established, and that ∇q holds if pp ever holds (see Lemma 9 below). However, it is not necessarily the case that if $beh.a$ holds in an interval, say Δ, $beh.a$ also holds for every sub-interval of Δ. For example, if the behaviour of an action changes after a certain length of time, then its behaviour over a longer interval will differ from its behaviour over a shorter one. Thus, we introduce the concept of a decomposable action.

Definition 8. *An action a* decomposes *iff for any intervals Δ and δ with $\delta \subseteq \Delta$, and trace tr, if $beh.a.\Delta.tr$, then $beh.a.\delta.tr$.*

Because teleo-reactive programs can be hierarchically nested, we often end up with a rely condition, say $\Box F$, which implies the guard of a higher-level action. For example, when proving properties of the grasp action (Fig. 3), we can use $\Box see\_can$ as a rely condition because see_can appears as a guard of fetch in robot.

Lemma 9 (Transitivity). *If* M *decomposes, then* $\{\Box F \wedge p \wedge \ell \geq D_1 + D_2\}$ M $\{\nabla G\}$ *holds if* $\{\Box F \wedge p \wedge \ell \geq D_1\}$ M $\{\nabla pp\}$ *and* $\{\Box F \wedge pp \wedge \ell \geq D_2\}$ M $\{\nabla G\}$ *hold for some state predicate pp.*

The next lemma states that proving ∇F is equivalent to proving ∇F holds in an interval that satisfies $\neg F$ (cf. [3]). Furthermore, ∇G holds under rely condition $F \,\mathbf{un}\, G$ iff $\circledcirc G$ holds under rely condition $\Box(F \wedge \neg G)$.

Lemma 10.
a. $\{r\}$ M $\{\nabla F\} = \{r \wedge \neg F\}$ M $\{\nabla F\}$
b. $\{r\}$ M $\{\nabla F\} = \{r \wedge \Box\neg F\}$ M $\{\nabla F\} = \{r \wedge \Box\neg F\}$ M $\{\circledcirc F\}$
c. $\{r \wedge (F \,\mathbf{un}\, G)\}$ M $\{\nabla G\} = \{r \wedge \Box(F \wedge \neg G)\}$ M $\{\circledcirc G\}$

The next lemma states that we can prove that a first guard is established by a program if the rest of the program establishes the guard.

Lemma 11 (Establish guard). *If* M $\widehat{=}$ $\langle c \rightarrow a\rangle \frown S$, *then* $\{r\}$ M $\{\nabla c\}$ *holds iff* $\{r\}\ S\ \{\nabla c\}$ *holds.*

The following theorem simplifies proofs of progress in goal-directed agents and facilitates construction of an appropriate rely condition. We are often required to prove properties of the form

$$\{\Box F \wedge p \wedge \ell \geq D\}\ \langle c \rightarrow a\rangle \frown S\ \{\nabla G\}$$

where p is some initial condition, F is an accumulation of guards and weak unless properties, D is the length of the interval and G is a temporal formula representing the goal of the first action, $c \rightarrow a$. The theorem generates the required proof obligations (7) and (8) given that F takes a particular form. Namely, F must ensure that the environment does not disable c unless the goal, G, is established.

Theorem 12 (Progression). *Suppose* $\mathsf{M} \mathrel{\widehat{=}} \langle c \rightarrow a \rangle \frown S$ *and* M *decomposes. Then*

$$\{\Box(F \wedge ((F' \wedge c)\ \mathbf{wu}\ G)) \wedge p \wedge \ell \geq D_1 + D_2\}\ \mathsf{M}\ \{\nabla G\} \tag{6}$$

holds provided $F \Rrightarrow F'$ *and*

$$\{\Box F \wedge p \wedge \neg c \wedge \ell \geq D_1\}\ S\ \{\nabla c\} \tag{7}$$

$$\{\Box(F \wedge c) \wedge \ell \geq D_2\}\ a\ \{\nabla G\} \tag{8}$$

Proof. Using Lemma 9 (transitivity), (6) holds if both of the following hold:

$$\{\Box(F \wedge ((F' \wedge c)\ \mathbf{wu}\ G)) \wedge p \wedge \ell \geq D_1\}\ \mathsf{M}\ \{\nabla c\} \qquad (prog1)$$

$$\{\Box(F \wedge ((F' \wedge c)\ \mathbf{wu}\ G)) \wedge c \wedge \ell \geq D_2\}\ \mathsf{M}\ \{\nabla G\} \qquad (prog2)$$

We now have the following calculations:

$(prog1)$
$\Leftarrow$ Corollary 4 (weaken/strengthen)
$\{\Box F \wedge p \wedge \ell \geq D_1\}\ \mathsf{M}\ \{\nabla c\}$
$=$ Lemma 11 (establish guard), Lemma 10 (a)
(7)

$(prog2)$
$\Leftarrow$ logic, Corollary 4 (weaken/strengthen)
$\{\Box F \wedge ((F' \wedge c)\ \mathbf{wu}\ G) \wedge c \wedge \ell \geq D_2\}\ \mathsf{M}\ \{\nabla G\}$
$\Leftarrow$ logic, Corollary 4 (weaken/strengthen): use $\Box F \wedge c$ and $F \Rrightarrow F'$
$\{\Box F \wedge ((F' \wedge c)\ \mathbf{un}\ G) \wedge \ell \geq D_2\}\ \mathsf{M}\ \{\nabla G\}$
$\Leftarrow$ $G \Rightarrow \nabla G$ and $\Box F \Rrightarrow \Box F'$
$\{\Box(F \wedge c) \wedge \ell \geq D_2\}\ \mathsf{M}\ \{\nabla G\}$
$\Leftarrow$ Lemma 7 (program reduction)
(8) □

Thus, instead of reasoning about the program as a whole, we reason over sub-programs a and S. This reasoning can be re-iterated over a (hierarchically) and S (iteratively).

At first glance, F' in formula $(F' \wedge c)\ \mathbf{wu}\ G$ may seem redundant due to the presence of F. However, in our verification, we compute the required rely condition of the overall program at each application of Theorem 12. In doing so, due to the hierarchical nesting, the context of the lower-level primitive actions may not always be present at the top-level, and hence, the final rely condition may become too strong. Thus, we use $(F' \wedge c)\ \mathbf{wu}\ G$ instead of just $c\ \mathbf{wu}\ G$. Similarly, the rely condition in (7) could be simplified to $\Box F \wedge p \wedge \ell \geq D_1$ using Lemma 10. However, if such a simplification is performed, we may lose the context $\neg c$ at the level of the primitive actions, i.e., when S is expanded.

4 Can clearing robot

In this section we prove a progress property for the example in Fig. 3. We specify the primitive actions in Section 4.1, specify the progress requirement in Section 4.2, and prove the progress property in Section 4.3.

4.1 Specification of primitive actions

To verify the correctness of the robot, we first specify the behaviour of each primitive durative action $pa \in \{\mathsf{rotate}, \mathsf{forward}, \mathsf{grasp}, \mathsf{ungrasp}\}$ using rely/guarantee specifications. To achieve this, each object on the table is associated with a vector, pos, that determines the position of the object in polar coordinates with a *magnitude* and an *angle*. Assume that addition and subtraction of vectors is defined in the standard manner. We can use the following definitions to determine whether or not the robot sees or holds the given can. We assume $robot.rot$ denotes the angle of rotation of the robot, $robot \approx can$ states that the robot is sufficiently close to can to be touching it, and TC is a set representing the cans that are on the table or being held by the robot.

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

That is, the robot can see the given object, obj, if 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 holds can if the robot sees and touches can and the holding sensor is on. The guards in Fig. 3, (which represent sensor values) satisfy the following equations, where $depot.pos$ is constant. Conditions see_can, $touching$, and $holding$ are sensed inputs whereas at_depot and see_depot are derived.

$$\begin{aligned} see\_can &= \exists can\colon TC \bullet sees.can & at\_depot &= (robot.pos = depot.pos) \\ touching &= \exists can\colon TC \bullet may\_hold.can & see\_depot &= sees.depot \\ holding &= \exists can\colon TC \bullet holds.can \end{aligned}$$

The rotate action causes the robot to rotate in position. We assume that the robot rotates at the rate of κ, and hence, for $D \in Time$, $\kappa \times D$ is the angle of rotation over D time units. Acceleration to (and deceleration from) κ are assumed to be instantaneous. Given that $\frac{d\,x}{d\,y}$ denotes the rate of change of x with respect to y, we have:

$$\mathsf{rotate} \mathrel{\widehat{=}} \{robot.rot\}\colon [\![true, \Box \left(\tfrac{d\,robot.rot}{d\,t} = \kappa\right)]\!] \tag{9}$$

That is, if the position of the robot is initially rp, then the robot remains at rp throughout execution of rotate. The robot rotates at a constant angular velocity through the interval.

Similarly, we use ξ for the speed of the robot and assume that acceleration to (and deceleration from) ξ is instantaneous. We assume that the frame of forward is $\{robot.pos\}$, and require that the following hold.

$$\{true\}\ \mathsf{forward}\ \{\Box(\tfrac{d\,robot.pos}{d\,t} = (\xi, robot.rot))\} \tag{10}$$

$$\{\Box holds.can\}\ \mathsf{forward}\ \{\Box(\tfrac{d\,can.pos}{d\,t} = (\xi, robot.rot))\} \tag{11}$$

We assume φ is the speed at which the fingers open and close and define grasp as follows.

$$\mathsf{grasp} \mathrel{\widehat{=}} \{gdist\}\colon \left[\!\left[true, \Box\left(\frac{d\,gdist}{d\,t} = (\mathsf{if}\ gdist = 0\ \mathsf{then}\ 0\ \mathsf{else}\ -\varphi)\right)\right]\!\right] \tag{12}$$

Thus, grasp causes the distance between the fingers of the robot gripper to decrease as long as the distance is greater than 0.

In our proof, ungrasp must modify an additional auxiliary variable dc which denotes the total number of cans that the robot has placed in the depot. Thus, the frame of ungrasp is $\{gdist, dc\}$. We require that ungrasp satisfies the following, where $k \in \mathbb{N}$.

$$\{true\}\ \mathsf{ungrasp}\ \left\{\Box\left(\frac{d\,gdist}{d\,t} = \begin{pmatrix}\mathsf{if}\ gdist = max\_gd \\ \mathsf{then}\ 0\ \mathsf{else}\ \varphi\end{pmatrix}\right)\right\} \tag{13}$$

$$\left\{\begin{array}{r}\Box(dc = k \wedge holds.can \wedge \\ at\_depot) \wedge \ell \geq D_u\end{array}\right\}\ \mathsf{ungrasp}\ \left\{\begin{array}{l}\circledcirc(\neg holds.can \Rightarrow \\ \quad (dc = k+1) \wedge can \notin TC)\end{array}\right\} \tag{14}$$

By (13), ungrasp causes $gdist$ to increase to a maximum of max_gd. By (14), if the number of cans in the depot is k, the robot is holding can at the depot and is performs an ungrasp action for an interval of length D_u or longer, then if in the next interval the robot is not holding can, dc must be incremented and can must no longer be in TC.

We assume that following holds, where $can \asymp can2$ states that can and $can2$ not overlapping, i.e.,

$$\Box(\forall can, can2\colon TC \bullet can \neq can2 \Rightarrow \neg(can \asymp can2))$$

4.2 Requirement specification

We define $can\_exists \mathrel{\widehat{=}} TC \neq \{\}$. Our progress requirement is that the following must hold for any $k \in \mathbb{N}$, some durative formula r, and some $D \in Time$.

$$\{\Box(dc = k \wedge r) \wedge can\_exists \wedge \ell \geq D\}\ \mathsf{robot}\ \{\nabla(dc = k+1)\} \tag{15}$$

That is, for any interval of length D, if the number of cans in the depot is k throughout the interval, and at the start of the interval there is a can on the table and the gripping finger width is adequate, then the number of cans in the depot eventually goes up by one. We also assume a rely condition r is true over the interval because it is typically not possible to prove (15) directly. For example, the environment may remove all the cans from the table, which falsifies can_exists without establishing $dc = k+1$. Application of Theorem 12 (progression) describes how the rely condition r may be instantiated to describe the assumptions under which (15) holds. We hence construct rely condition r as the verification progresses.

4.3 Proof of (15)

We let $D = D_1 + D_2$ and define:

$$r \mathrel{\widehat{=}} r_1 \wedge r_2 \wedge (holding\ \mathbf{wu}\ dc = k+1) \tag{16}$$

then apply Theorem 12 (progression) where F is instantiated to $r_1 \wedge r_2 \wedge dc = k$ and F' is instantiated to $true$. Then, we use Corollary 4 (weaken rely) to remove conjunct $dc = k \wedge r_2$ from the first rely condition and r_1 from the second, and hence obtain the following proof obligations.

$$\{\Box r_1 \wedge can\_exists \wedge \neg holding \wedge \ell \geq D_1\} \ \mathsf{collect} \ \{\nabla holding\} \tag{17}$$

$$\{\Box(dc = k \wedge r_2 \wedge holding) \wedge \ell \geq D_2\} \ \mathsf{deliver} \ \{\nabla(dc = k + 1)\} \tag{18}$$

By (17), we must show that collect is guaranteed to establish $holding$. Using Corollary 4, we have removed $\Box(dc = k)$ from the rely condition of (17) because progress in collect is not affected by the number of cans in the depot. Condition (18) is an instantiation of (8) which states that deliver is guaranteed to increment dc, given that $dc = k$ and the robot is holding a can throughout the interval. By (16), if the robot is holding a can at the start of the interval, then it will continue to do so, i.e., the environment cannot remove the can from the robot's grasp unless $dc = k + 1$ is established.

Proof of (17). Using Theorem 12 (progression), we obtain the following proof obligations where $D_1 = D_{1.1} + D_{1.2}$ and

$$r_1 \mathrel{\widehat{=}} r_{1.1} \wedge r_{1.2} \wedge (see\_can \ \mathbf{wu} \ holding) \tag{19}$$

We define $nhsc \mathrel{\widehat{=}} \neg holding \wedge \neg see\_can$.

$$\{\Box r_{1.1} \wedge can\_exists \wedge nhsc \wedge \ell \geq D_{1.1}\} \ \mathsf{rotate} \ \{\nabla see\_can\} \tag{20}$$

$$\{\Box(r_{1.2} \wedge see\_can) \wedge \ell \geq D_{1.2}\} \ \mathsf{fetch} \ \{\nabla holding\} \tag{21}$$

Thus, rotate must enable fetch, which in turn must ensure that the robot will eventually hold a can. Furthermore, the environment must be such that if the robot can see a can, then the robot must continue to see a can unless it holds a can.

Proof of (20). The proof of this property requires that one of the cans on the table does not move. Otherwise the environment may move the cans in a manner that prevents the rotate action from ever seeing a can without falsifying $can\_exists \wedge nhsc$. Thus, we introduce:

$$\begin{aligned} r_{1.1} \mathrel{\widehat{=}} (&can\_exists \wedge nhsc \Rightarrow \\ &\exists can\colon TC \bullet can.pos = cp \wedge ((can.pos = cp \wedge nhsc) \ \mathbf{un} \ see\_can)) \end{aligned} \tag{22}$$

Condition (22) ensures that if a can is on the table, the robot is not holding anything, and robot does not see a can, then there is a can that does not move unless the robot sees a can. Note that the can that the robot sees may be different from the can that is not moving. Condition (22) is not a strong enough assumption because it does not disallow cans from moving after robot has seen a can. In particular, if the can the robot is currently seeing moves, then the robot may not make progress because it may not reach that can. We address this issue in (28) below as part of the proof of fetching a can. Returning to the proof of (20), we obtain:

$$\begin{aligned} & (20) \\ \Leftarrow \quad & \text{Lemma 3 (weaken/strengthen) and (22)} \end{aligned}$$

$$\left\{\begin{array}{l}\exists can\colon TC \bullet can.pos = cp \wedge \ell \geq D_{1.1} \wedge \\ \quad ((can.pos = cp \wedge nhsc)\ \mathbf{un}\ see\_can)\end{array}\right\} \mathsf{rotate}\ \{\nabla see\_can\}$$

$\Leftarrow$ Lemma 5

$$\forall can\colon TC \bullet \left\{\begin{array}{l}can.pos = cp \wedge \ell \geq D_{1.1} \wedge \\ ((can.pos = cp \wedge nhsc)\ \mathbf{un}\ see\_can)\end{array}\right\} \mathsf{rotate}\ \{\nabla see\_can\}$$

$=$ Lemma 10

$$\forall can\colon TC \bullet \{\Box(can.pos = cp \wedge nhsc) \wedge \ell \geq D_{1.1}\}\ \mathsf{rotate}\ \{\obar\, see\_can\}$$

$\Leftarrow$ logic

(9)

The rely condition of the second last line in the proof above states that for the interval under consideration, the robot is not holding or seeing a can, the position of each can does not change, and the rotate action is executing. Given that this interval is of a certain length, see_can must become true at the end of the interval. This clearly holds by (9) which guarantees the robot does not change position, but increases the value of $robot.rot$. Hence $(can.pos - robot.pos).angle \ \mathsf{mod}\ 2\pi = robot.rot \ \mathsf{mod}\ 2\pi$ will eventually hold, which implies see_can.

Proof of (21). We define $gOK \mathrel{\widehat{=}} touching \wedge (gdist > can\_width)$, $r_{1.2} \mathrel{\widehat{=}} r_{1.2.1} \wedge (gOK\ \mathbf{wu}\ holding)$, $D_{1.2} \mathrel{\widehat{=}} D_{1.2.1} + D_{1.2.2}$, and $\mathsf{S} \mathrel{\widehat{=}} \langle \mathsf{fetch}, \mathsf{ungrasp}\rangle$, then apply Theorem 12 (progression) and Corollary 4 (weaken strengthen) to obtain the following.

$$\{\Box(r_{1.2.1} \wedge see\_can) \wedge \ell \geq D_{1.2.1}\} \quad \mathsf{S} \quad \{\nabla gOK\} \tag{23}$$

$$\{\Box(see\_can \wedge \neg holding \wedge gOK) \wedge \ell \geq D_{1.2.2}\}\ \mathsf{grasp}\ \{\nabla holding\} \tag{24}$$

Condition (24) is an immediate consequence of (12), which describes the behaviour of primitive action grasp. For condition (23) we apply Theorem 12 (progression) with $r_{1.2.1} \mathrel{\widehat{=}} rr \wedge ((gdist > can\_width)\ \mathbf{wu}\ gOK)$ and $D_{1.2.1} \mathrel{\widehat{=}} D_{1.2.1.1} + D_{1.2.1.2}$. Then we apply Corollary 4 (weaken strengthen) to obtain:

$$\{\ell \geq D_{1.2.1.1}\}\ \mathsf{ungrasp}\ \{\nabla(gdist > can\_width)\} \tag{25}$$

$$\{\Box(rr \wedge see\_can \wedge gdist > can\_width) \wedge \ell \geq D_{1.2.1.2}\}\ \mathsf{forward}\ \{\nabla gOK\} \tag{26}$$

Condition (25) is an immediate consequence of (13), while (26) can be simplified using Lemma 10 (b) to obtain

$$\left\{\begin{array}{l}\Box(rr \wedge see\_can \wedge (gdist > can\_width) \wedge \\ \quad \neg touching) \wedge \ell \geq D_{1.2.1}\end{array}\right\} \mathsf{forward}\ \{\obar\, gOK\} \tag{27}$$

We require that any can that is being fetched cannot be moving, otherwise, the forward action would not cause the robot to touch a can. Thus, we define:

$$rr \mathrel{\widehat{=}} \forall can\colon TC \bullet sees.can \Rightarrow (can.pos = cp\ \mathbf{wu}\ gOK) \tag{28}$$

which, together with the definition of $sees\_can$ and (10) allows us to prove (27).

Proof of (18). Defining $r_2 \mathrel{\widehat{=}} r_{2.1} \wedge (at\_depot\ \mathbf{wu}\ dc = k+1)$ and $D_2 \mathrel{\widehat{=}} D_{2.1} + D_{2.2}$, then using Theorem 12 (progression) and Corollary 4 (weaken/strengthen) we obtain:

$$\{\Box(r_{2.1} \wedge holding) \wedge \neg at\_depot \wedge \ell \geq D_{2.1}\}\ \mathsf{go\_depot}\ \{\nabla at\_depot\} \tag{29}$$

$$\{\Box(dc = k \wedge holding \wedge at\_depot) \wedge \ell \geq D_{2.2}\}\ \mathsf{ungrasp}\ \{\obar(dc = k+1)\} \tag{30}$$

Condition (30) is an immediate consequence of (14) provided $D_{2.2} \geq D_u$.

Proof of (29). We define $r_{2.1} \mathrel{\widehat{=}} (holding \wedge see\_depot)\ \mathbf{wu}\ at\_depot$ and using Theorem 12 (progression) and Corollary 4 (weaken/strengthen) we obtain:

$$\{\Box holding \wedge \neg at\_depot \wedge \neg see\_depot \wedge \ell \geq D_{2.1.1}\}\ \mathsf{rotate}\ \{\nabla see\_depot\} \quad (31)$$

$$\{\Box(holding \wedge see\_depot) \wedge \ell \geq D_{2.1.2}\}\ \mathsf{forward}\ \{\nabla at\_depot\} \quad (32)$$

Both (31) and (32) are consequences of the primitive action specifications together with the fact that the position of the depot is constant.

This completes the proof of progress as formalised by (15). After simplification, the final rely condition is:

$$\begin{array}{l}
(\ell \geq D_{1.1} + D_{1.2.1} + D_{1.2.2} + D_{2.1.1} + D_{2.1.2} + D_{2.2}) \wedge can_exists \wedge \\
\Box(dc = k \wedge (see_can\ \mathbf{wu}\ holding) \wedge (gOK\ \mathbf{wu}\ holding) \wedge \\
\quad (can_exists \wedge nhsc \Rightarrow \\
\qquad \exists can\colon TC \bullet can.pos = cp \wedge ((can.pos = cp \wedge nhsc)\ \mathbf{un}\ see_can)) \wedge \\
\quad (\forall can\colon TC \bullet sees.can \Rightarrow (can.pos = cp\ \mathbf{wu}\ gOK)) \wedge \\
\quad (holding \Rightarrow \Box holding) \wedge (at_depot \Rightarrow \Box at_depot) \wedge \\
\quad ((holding \wedge see_depot)\ \mathbf{wu}\ at_depot))
\end{array}$$

Note that $holding\ \mathbf{wu}\ dc = k + 1$ from (16) in has been simplified to $holding \Rightarrow \Box holding$ in the above formula because $\Box(dc = k)$ holds. (Similarly, $at\_depot\ \mathbf{wu}\ dc = k+1$.) Furthermore, the ungrasp action guarantees $gdist > can\_width$. Thus, we conclude that for any interval (of adequate length) that satisfies the the condition above, execution of the robot is such that $\nabla(dc = k + 1)$ will hold.

5 Conclusion

In this paper we have formalised the semantics of teleo-reactive programs. We have defined durative temporal logic, which is a temporal logic for reasoning about continuous intervals. 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. We are able to use the lemmas in Sections 3.1 and 3.2 to simplify the proof obligations.

Our example assumes an idealised scenario where several physical constraints are simplified. For instance, we assume acceleration is instantaneous, 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 verification is to demonstrate applicability of our logic for the verification of goal-based agents. Removing our idealised assumptions would make the verification more complicated, but a verification is nevertheless possible.

We regard verification of a larger, more complicated example to be future work. To this end, we are currently developing mechanised proofs and have a Prolog program that automatically constructs the required rely condition via repeated application of Theorem 12 (progression).

Acknowledgements. We would like to thank Keith Clark and Kirsten Winter for helpful discussions on this paper. This research is supported by Australian Research Council (ARC) Discovery Grant DP0558408 and The University of Queensland's New Staff Start-up Research Fund.

References

1. R. J. Back. Refinement of parallel and reactive programs. In M. Broy, editor, *Lecture Notes For the Summer School on Program Design Calculi*, pages 73–92. Springer-Verlag, 1993.
2. R. J. Back, L. Petre, and I. Porres. Generalizing action systems to hybrid systems. In M. Joseph, editor, *FTRTFT*, volume 1926 of *LNCS*, pages 202–213. Springer, 2000.
3. B. Dongol and A. J. Mooij. Progress in deriving concurrent programs: Emphasizing the role of stable guards. In Uustalu [16], pages 140–161.
4. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072. Elsevier Science Publishers, 1990.
5. S. Fritsch, A. Senart, D. C. Schmidt, and S. Clarke. Time-bounded adaptation for automotive system software. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 571–580, New York, NY, USA, 2008. ACM.
6. G. Gubisch, G. Steinbauer, M. Weiglhofer, and F. Wotawa. A teleo-reactive architecture for fast, reactive and robust control of mobile robots. In *IEA/AIE '08: Proceedings of the 21st International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 541–550, Berlin, Heidelberg, 2008. Springer-Verlag.
7. I. J. Hayes. Towards reasoning about teleo-reactive programs for robust real-time systems. In *SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*, pages 87–94, New York, NY, USA, 2008. ACM.
8. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.
9. L. Lamport. Hybrid systems in TLA$^+$. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 77–102. Springer, 1992.
10. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
11. C. Lewerentz and T. Lindner, editors. *Formal Development of Reactive Systems - Case Study Production Cell*, volume 891 of *LNCS*. Springer, 1995.
12. Z. Manna and A. Pnueli. *Temporal Verification of Reactive and Concurrent Systems: Specification*. Springer-Verlag New York, Inc., 1992.
13. L. Meinicke and I. J. Hayes. Continuous action system refinement. In Uustalu [16], pages 316–337.
14. F. Nafz, F. Ortmeier, H. Seebach, J. P. Steghöfer, and W. Reif. A universal self-organization mechanism for role-based organic computing systems. In W. Reif, G. Wang, and J. Indulska, editors, *ATC*, volume 5586 of *LNCS*, pages 17–31. Springer, 2009.
15. N. J. Nilsson. Teleo-reactive programs and the triple-tower architecture. *Electronic Transactions on Artificial Intelligence*, 5:99–110, 2001.
16. T. Uustalu, editor. *Proceedings of the 8th International Conference on Mathematics of Program Construction*, volume 4014 of *LNCS*. Springer, 2006.
17. J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
18. C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. EATCS: Monographs in Theoretical Computer Science. Springer, 2004.