THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Reasoning About Teleo-Reactive Programs Under Parallel Composition

Brijesh Dongol
Ian J. Hayes

April 2011

Technical Report SSE-2011-01

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 Teleo-Reactive Programs Under Parallel Composition

Brijesh Dongol and Ian J. Hayes

April 27, 2011

Abstract

The teleo-reactive programming model is a high-level approach to implementing real-time controllers that react dynamically to changes in their environment. Teleo-reactive actions can be hierarchically nested, which facilitates abstraction from lower-level details. Furthermore, teleo-reactive programs can be composed using renaming, hiding, and parallelism to form new programs. In this paper, we present a framework for reasoning about safety, progress, and real-time properties of teleo-reactive programs under program composition. We use a logic that extends the duration calculus to formalise the semantics of teleo-reactive programs and to reason about their properties. We present rely/guarantee style specifications to allow compositional proofs and we consider an application of our theory by verifying a real-time controller for an industrial press.

1 Introduction

With the increasing sophistication of real-time safety-critical systems, it is important to develop more sophisticated provably correct programming methodologies. For example, development of provably correct real-time controllers for robot motion has been identified to be a "grand challenge" of robotics [4]. Teleo-reactive programs [20] are high-level programs that have been identified to be a good candidate for developing reactive real-time software [10, 7], presenting a fundamentally different approach to programming in comparison to state machine style methods.

Each action of a teleo-reactive program is *durative*, i.e., occurs over an interval of time. Durative actions can describe rates of change of state variables over time as opposed to explicitly changing the values of these state variables. Teleo-reactive programs naturally support hierarchical nesting [7, 20] which allows details of the lower-level programs to be developed at a later stage. Furthermore, several teleo-reactive programs may execute in parallel [20], with individual programs controlling different aspects of a complex system.

In this paper, we develop techniques for reasoning about teleo-reactive programs under parallel composition. We also consider renaming and hiding and present some special cases of parallel composition (pipelines and simple parallelism). We use a logic called durative temporal logic [7], which is based on the duration calculus [22] and linear temporal logic [17]. We use rely/guarantee style reasoning to allow compositional proofs. Our framework allows reasoning about safety, progress and real-time properties of teleo-reactive programs.

1.1 Example

To highlight the differences between teleo-reactive programs and state-machine frameworks, we consider a teleo-reactive program for controlling a lift that moves up to collect objects and delivers them to the bottom.

$$\mathsf{Lift} \mathrel{\widehat{=}} \left\langle \begin{array}{c} \mathit{door\_closed} \to \mathsf{runLift}, \\ \mathit{true} \to \mathit{Nil} \end{array} \right\rangle \qquad \mathsf{runLift} \mathrel{\widehat{=}} \left\langle \begin{array}{r} \mathit{lift\_full} \land \neg \mathit{bottom} \to \mathit{Lower}, \\ \mathit{lift\_empty} \land \neg \mathit{top} \to \mathit{Raise}, \\ \mathit{true} \to \mathit{Nil} \end{array} \right\rangle$$

The main program Lift executes program $\mathsf{runLift}$ in any interval in which the door is closed, i.e., *door_closed* holds and executes *Nil* (which does nothing) otherwise. Program $\mathsf{runLift}$ lowers the lift if it is full and not at the bottom, raises the lift if it is empty and not at the top, and does nothing otherwise.

In an execution of a non-empty sequence of guarded programs, the guard of each program in the sequence is continuously evaluated, and the first enabled program from the sequence is executed. For example, in program Lift, action $\mathsf{runLift}$ is executed while *door_closed* holds and *Nil* (which does nothing) is executed otherwise. If *door_closed* ever becomes false while $\mathsf{runLift}$ is executing, then $\mathsf{runLift}$ stops and *Nil* starts executing. Thus, Lift is equivalent to $\langle door\_closed \rightarrow \mathsf{runLift}, \neg door\_closed \rightarrow Nil\rangle$. Teleo-reactive programs also naturally support hierarchical composition, e.g., the $\mathsf{runLift}$ program executes within the context of the *door_closed* guard, i.e., each guard in $\mathsf{runLift}$ implicitly has *door_closed* as a conjunct.

Teleo-reactive programs are reactive, i.e., execute over a dynamically changing environment, and hence, the value of *door_closed* may be controlled (i.e., modified) by the environment of Lift. Furthermore, unlike state-machine like models such as hybrid automata, the guarded actions of teleo-reactive programs are durative, i.e., each guarded action continues to execute over an interval in which its guard holds. For example, the semantics of the behaviour of *Lower* describes the rate behaviour of the lift while *Lower* is executing. This is in contrast to hybrid systems that would use a pair of assignments, say $state := lower$ and $state := nil$ lower and stop lowering the lift, and/or $lift\_speed := x$ to set the rate at which lift is lowered.

Teleo-reactive programs are often used to implement goal-directed agents [20]. That is, we structure a program $T = \langle c \rightarrow \mathsf{M}\rangle \frown S$ so that execution of S achieves subgoals that are required for c to hold, which in turn enables M to achieve its goal. In the $\mathsf{runLift}$ program above, the overall goal of the lift is to lower objects to the bottom and hence, the *Lower* action is the first action in the sequence. The *Raise* action appears next because the lift must go to the the top to receive objects, i.e., *Raise* achieves the subgoal of establishing *lift_full*.

1.2 Related work

This paper is concerned with a logic for composing teleo-reactive programs. As far as we are aware, such a logic thus far not been developed, although there are a number of formalisms available for reasoning about hybrid and continuous systems. Many of these techniques extend existing discrete state-based formalisms to a hybrid model, e.g., continuous action systems [3, 18], hybrid action systems [21], TLA$^+$ [14], timed automata [1]. Here, variables are considered to be of type $Time \rightarrow Val$ (where $Time \mathrel{\widehat{=}} \mathbb{R}$), to allow continuous behaviour to be described. Parallel composition of teleo-reactive programs is simpler than these methods because synchronisation of actions is not required.

Compositional verification of real-time systems is clearly desirable, and almost any new formalism encompasses some sort of compositional technique [8]. However, some existing techniques require an explicit clock to be implemented or assume an interleaving model of concurrency [23, 11], while others assume a synchronous execution [2]. These restrictions do not suit the teleo-reactive framework. Furia et al. present a compositional real-time framework that does not make any assumptions on the model of concurrency, however, their model requires the guarantee continue to hold past the interval in which the rely condition holds [8].

A logic for reasoning about a single-process teleo-reactive program has been developed [7]. In this paper, we expand the theory and present techniques for reasoning about teleo-reactive programs that consist of communicating parallel processes. Our techniques allow properties of the subprograms to be used, i.e., compositional reasoning, when reasoning about the system built from them.

Our real-time logic is most influenced by the duration calculus [22] but tailored to suit the teleo-reactive programming model, e.g., we consider both open and closed intervals. We do not use the duration calculus directly because its rules focus on lower-level reasoning and on relationships between intervals.

This paper is organised as follows. In Section 2 we present our real-time logic and in Section 3 we present the syntax and semantics teleo-reactive programs. We present our rules for reasoning about teleo-reactive programs in Section 4 and in Section 5 we present a case study by verifying an abridged version of the production cell.

2 A real time framework

In Section 2.1, we present some preliminary theory on intervals, streams and predicates. In Section 2.2, we present a theory for reasoning over partitions of intervals.

2.1 Preliminaries

Interval predicates An interval is a contiguous subset of *Time* (represented by real numbers $\mathbb{R}$). Intervals may either be open or closed at either end and may also be infinite. An interval has type

$$Interval \mathrel{\widehat{=}} \left\{\Delta \subseteq \mathbb{R} \,\middle|\, \Delta \neq \{\} \wedge \forall t, t' \in \Delta \bullet t < t' \Rightarrow \forall t'' \colon \mathbb{R} \bullet t < t'' < t' \Rightarrow t'' \in \Delta \right\}$$

Thus, if t and t' are in the interval Δ, then all real numbers between t and t' are also in Δ. For an interval $\Delta \in Interval$, we let $lub.\Delta$ and $glb.\Delta$ denote the least upper and greatest lower bounds of Δ, respectively where '.' denotes function application. We use $\ell.\Delta$ (equal to $lub.\Delta - glb.\Delta$) denote the length of Δ. For intervals $\Delta, \Delta' \in Interval$, we define the *adjoins* relation between Δ and Δ' as follows:

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

That is, $\Delta \propto \Delta'$ states that Δ' is an interval that immediately follows Δ.

We define a *state space* as $\Sigma_V \mathrel{\widehat{=}} V \to Val$ where $V \subseteq Var$ is a set of variables and *Val* a set of values. We leave out the subscript if V is clear from the context. A *predicate* over a type X is given by $\mathcal{P}X \mathrel{\widehat{=}} X \to \mathbb{B}$, a *state* is a member of Σ, and a *state predicate* is a member of $\mathcal{P}\Sigma$. The (real-time) stream is given by $Stream_V \mathrel{\widehat{=}} Time \to \Sigma_V$ which is a total function from times to states with variables V. A *stream predicate* is a member of $\mathcal{P}Stream_V$ and an *interval predicate* is a member of the set $IntvPred_V \mathrel{\widehat{=}} Interval \to \mathcal{P}Stream_V$. Interval predicates allow us to reason about the behaviour of a stream with respect to a given interval. We let $vars.c$ and $vars.p$ denote the sets of all variables V that may occur free in $c \in \mathcal{P}\Sigma_V$ and $p \in IntvPred_V$.

The boolean operators may be lifted pointwise to state and interval predicates, e.g., $(p_1 \wedge p_2).\Delta.tr = (p_1.\Delta.tr \wedge p_2.\Delta.tr)$ for interval predicates p_1 and p_2. We define some further notation for stream predicates sp_1 and sp_2:

$$\begin{array}{rcl} (sp_1 \Rrightarrow sp_2) & \widehat{=} & \forall tr \colon Stream \bullet sp_1.tr \Rightarrow sp_2.tr \\ (p_1 \Rrightarrow p_2) & \widehat{=} & \forall \Delta \colon Interval \bullet p_1.\Delta \Rrightarrow p_2.\Delta \end{array}$$

'$\Lleftarrow$' and '$\equiv$' are similarly defined with '$\Rightarrow$' replaced by '$\Leftarrow$' and '$=$', respectively.

We let $\lim\limits_{x \to a^-} f.x$ and $\lim\limits_{x \to a^+} f.x$ denote the limit of $f.x$ from the left and right, respectively. To ensure that the limit is well-defined, we assume that each variable $v \in V$ is piecewise continuous in $s \in Stream_V$ [9]. For an expression $e \in \Sigma \to Val$, interval $\Delta \in Interval$ and stream $s \in Stream$, we define:

$$\begin{array}{rcl} \overrightarrow{e}.\Delta.s & \widehat{=} & \lim\limits_{t \to lub.\Delta^-} e.s_t \\ \overleftarrow{e}.\Delta.s & \widehat{=} & \lim\limits_{t \to glb.\Delta^+} e.s_t \\ (\downarrow e).\Delta & \widehat{=} & \exists \Delta' \colon Interval \bullet (\Delta' \propto \Delta) \wedge \overrightarrow{e}.\Delta' \\ (\uparrow e).\Delta & \widehat{=} & \exists \Delta' \colon Interval \bullet (\Delta \propto \Delta') \wedge \overleftarrow{e}.\Delta' \end{array}$$

Thus, $\overleftarrow{e}$ and $\overrightarrow{e}$ return the value of e at the *start* and *end* of the given interval, respectively, while $\downarrow e$ and $\uparrow e$ denote the value of e *before* and *after* the given interval, respectively. Note that e may be a state predicate, in which case the operators above evaluate to a boolean. For a state predicate c, the *everywhere* and *sometime* operators are defined as follows:

$$\begin{array}{rcl} (\circledast c).\Delta.s & \widehat{=} & \forall t \colon \Delta \bullet c.s_t \\ (\boxdot c).\Delta.s & \widehat{=} & \exists t \colon \Delta \bullet c.s_t \end{array}$$

Thus, $\circledast c$ and $\boxdot c$ hold iff c holds at every and some time in the given interval, respectively. We define the *chop* and *always* in a similar manner to the duration calculus [22]. Given interval predicates $p, p_1, p_2 \in IntvPred$ and interval $\Delta \in \mathit{Interval}$ we define:

$$\begin{aligned}
(p_1 \,;\, p_2).\Delta &\mathrel{\widehat{=}} \exists \Delta_1, \Delta_2 \colon \mathit{Interval} \bullet (\Delta_1 \propto \Delta_2) \wedge (\Delta = \Delta_1 \cup \Delta_2) \wedge p_1.\Delta_1 \wedge p_2.\Delta_2 \\
(\Box p).\Delta &\mathrel{\widehat{=}} \forall \Delta' \colon \mathit{Interval} \bullet \Delta' \subseteq \Delta \Rightarrow p.\Delta' \\
(\bigcirc p).\Delta &\mathrel{\widehat{=}} \exists \Delta' \colon \mathit{Interval} \bullet (\Delta \propto \Delta') \wedge p.\Delta'
\end{aligned}$$

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. The *everywhere* operator, $\Box$, states that the given interval predicate to hold over all subintervals of the given interval. We define the following shorthand notation:

$$p_1 : p_2 \mathrel{\widehat{=}} p_1 \vee (p_1 \,;\, p_2) \tag{1}$$
$$\Diamond p \mathrel{\widehat{=}} \neg \Box \neg p \tag{2}$$
$$\nabla p \mathrel{\widehat{=}} \Diamond p \vee \bigcirc p \tag{3}$$
$$p_1 \,\mathbf{un}\, p_2 \mathrel{\widehat{=}} p_2 \vee (\Box p_1 ;\, p_2) \vee (\Box p_1 \wedge \bigcirc (p_1 \vee p_2)) \tag{4}$$
$$p_1 \,\mathbf{wu}\, p_2 \mathrel{\widehat{=}} p_1 \Rightarrow (p_1 \,\mathbf{un}\, p_2) \tag{5}$$

The *weak chop* $(p_1 : p_2).\Delta$ holds iff p_1 holds over Δ or if $(p_1 \,;\, p_2).\Delta$ holds, $\Diamond p$ states that p holds in some subinterval of the given interval, ∇p states that p holds sometime within or immediately after the given interval, $p_1 \,\mathbf{un}\, p_2$ states that p_1 holds *unless* p_2 holds and $p_1 \mathbf{wu} p_2$ is the *weak unless* operator, which only requires $p_1 \,\mathbf{un}\, p_2$ to hold if p_1 holds.

Because an interval predicate has access to entire stream it may mention properties of the stream outside the given interval. As an extreme example, we define

$$(\amalg p).\Delta.s \mathrel{\widehat{=}} p.\mathit{Time}.s$$

which states that p hold over all time in s, i.e., $(\amalg p).\Delta$ ignores the given interval Δ.

Two adjacent intervals do not overlap at any point. Because our expressions are only piecewise continuous, we must use $\downarrow$ to link the last value of an expression in the previous interval to the first value in the current interval. In particular, we use $\downarrow$ to define invariance of a state predicate.

Definition 1 *A state predicate c is* invariant *over an interval Δ iff $(inv.c).\Delta$ holds, where*

$$\mathit{inv}.c \mathrel{\widehat{=}} \downarrow c \Rightarrow \circledast c$$

Thus, $\mathit{inv}.c$ holds iff c continues to hold within the given interval provided that $\downarrow c$ holds. Using inv, we define stability of a variable v and a set of variables V as follows:

$$\mathit{st}.v \mathrel{\widehat{=}} \exists k \bullet \mathit{inv}.(v = k) \tag{6}$$
$$\mathit{st}.V \mathrel{\widehat{=}} \forall v \colon V \bullet \mathit{st}.v \tag{7}$$

Thus, if the value of v is k immediately before the given interval, then the value of v remains k for the whole of the interval. A set of variables V is stable if each variable in V is stable.

2.2 Partitions, splits and joins

We often reason about a large interval by reasoning about its subintervals. It is particularly useful to consider a *partition* of an interval. We use $\mathrm{seq}\,.X$ to denote a possibly infinite sequence with elements of type X. A sequence can be explicitly defined using angle brackets, '$\langle$' and '$\rangle$', and '$\frown$' is the sequence concatenation operator. For a sequence of sets σ, we define we define $\bigcup \sigma \mathrel{\widehat{=}} \bigcup_{i:\mathrm{dom}\,.\sigma} \sigma_i$.

Definition 2 (Partition) *A* partition *of an interval $\Delta \in \mathit{Interval}$ is given by*

$$\mathit{part}.\Delta \mathrel{\widehat{=}} \{z \colon \mathrm{seq}\,.\mathit{Interval} \mid (\Delta = \bigcup z) \wedge (\forall i \colon \mathrm{dom}\,.z - \{0\} \bullet z_{i-1} \propto z_i)\}$$

A non-Zeno partition *of an Δ is given by*

$$\mathit{NZpart}.\Delta \mathrel{\widehat{=}} \{z \colon \mathit{part}.\Delta \mid (\mathrm{dom}\,.z = \mathbb{N}) \Rightarrow (\ell.\Delta = \infty)\}$$

Definition 3 (Alternates) *For a state predicate c, interval $\Delta \in Interval$ and a partition $\delta \in part.\Delta$, we define*

$$\begin{aligned} alt.c.\delta \quad \widehat{=} \quad & \forall i\colon \mathrm{dom}\,.\delta \bullet \; ((\boxast c).\delta_i \wedge (i+1 \in \mathrm{dom}\,.\delta) \Rightarrow (\boxast \neg c).\delta_{i+1}) \wedge \\ & ((\boxast \neg c).\delta_i \wedge (i+1 \in \mathrm{dom}\,.\delta) \Rightarrow (\boxast c).\delta_{i+1}) \end{aligned}$$

Definition 4 (Non-Zeno) *A state predicate c is* non-Zeno *in Δ iff there exists a $\delta \in NZpart.\Delta$ such that $alt.c.\delta$ holds and we say c is* non-Zeno *iff c is non-Zeno in every interval $\Delta \in Interval$.*

Definition 5 *Suppose p is an interval predicate. We say*

1. *p* joins *in Δ iff $(\forall \delta\colon NZpart.\Delta \bullet \forall i\colon \mathrm{dom}\,.\delta \bullet p.\delta_i) \Rrightarrow p.\Delta$.*
2. *p* splits *in Δ iff $p.\Delta \Rrightarrow \forall \delta\colon NZpart.\Delta \bullet (\forall i\colon \mathrm{dom}\,.\delta \bullet p.\delta_i)$.*

We say p joins *and p* splits *iff p joins in Δ and p splits in Δ, respectively for any arbitrary interval Δ.*

If p joins and holds over all intervals within an arbitrary partition of Δ, then p is guaranteed to hold over Δ. Conversely, if p splits and $p.\Delta$ holds, then p may be distributed over any partition of Δ. Note that if p joins then $(p\,;\; p) \Rrightarrow p$ and if p splits then $p \Rrightarrow \Box p$.

Lemma 1 *For any state predicate c, interval predicate $inv.c$ both joins and splits.*

The next lemma allows us to perform case analysis to prove formulae of the form $p_1 \Rrightarrow p_2$, provided that the case analysis is performed on a non-Zeno state predicate.

Lemma 2 (Split) *If p_1 splits and p_2 joins, then $p_1 \Rrightarrow p_2$ holds provided there exists a non-Zeno state predicate c and both of the following hold:*

$$p_1 \wedge \boxast c \quad \Rrightarrow \quad p_2 \tag{8}$$

$$p_1 \wedge \boxast \neg c \quad \Rrightarrow \quad p_2 \tag{9}$$

Proof 1 *For an arbitrary interval $\Delta \in Interval$,*

$$\begin{array}{ll} & p_1.\Delta \\ \Rrightarrow & \quad \textit{c is non-Zeno} \\ & p_1.\Delta \wedge \exists \delta\colon NZpart.\Delta \bullet alt.c.\delta \\ \Rrightarrow & \quad \textit{Definition 5, } p_1 \textit{ splits} \\ & \exists \delta\colon NZpart.\Delta \bullet alt.c.\delta \wedge \forall i\colon \mathrm{dom}\,.\delta \bullet p_1.\delta_i \\ \Rrightarrow & \quad \textit{(8) and (9)} \\ & \exists \delta\colon NZpart.\Delta \bullet \forall i\colon \mathrm{dom}\,.\delta \bullet p_2.\delta_i \\ \Rrightarrow & \quad \textit{Definition 5, } p_2 \textit{ joins} \\ & p_2.\Delta \end{array}$$

□

We may use transitivity to split proofs of progress properties. The proof for this lemma may be found in [7].

Lemma 3 (Transitivity) *Suppose p_1 and p_2 are interval predicates, c is a state predicate, p_1 splits, and $0 < \epsilon_1, \epsilon_2 \in Time$. Then*

$$p_1 \wedge \overleftarrow{c} \wedge (\ell \geq \epsilon_1 + \epsilon_2) \Rrightarrow \nabla p_2$$

holds provided that for some state predicate c', both of the following hold:

$$p_1 \wedge \overleftarrow{c} \wedge (\ell \geq \epsilon_1) \quad \Rrightarrow \quad \nabla \overleftarrow{d} \tag{10}$$

$$p_1 \wedge \overleftarrow{d} \wedge (\ell \geq \epsilon_2) \quad \Rrightarrow \quad \nabla p_2 \tag{11}$$

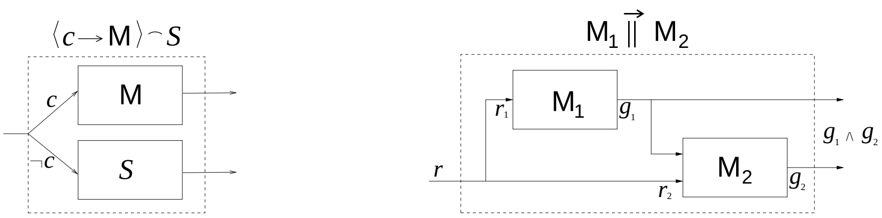

Figure 1: Guarded sequence and parallel composition

3 Teleo-reactive programs with parallel composition

In this section, we formalise the syntax and semantics of teleo-reactive programs under various forms for composition and present a rely/guarantee style framework for reasoning about their properties. We present the abstract syntax of teleo-reactive programs in Section 3.1 and provide their semantics in Section 3.2.

3.1 Syntax

Definition 6 *The abstract syntax of a* teleo-reactive program *is given by P below.*

$$\begin{array}{rcl} GP & ::= & c \to P \\ P & ::= & O\colon [\![r, g]\!] \;\mid\; \mathrm{seq}\,.GP \;\mid\; P \overrightarrow{\|} P \end{array}$$

An action $O\colon [\![r, g]\!]$ consists of a set of input variables, I, a *rely* condition, r, a *guarantee* condition, g, and a set of output variables, O. A guarded program $c \to \mathsf{M}$ consists of a guard c and a program M. A basic program may either be an action, a sequence of guarded programs or formed using the parallel composition operator (cf. Fig. 1). *Parallel composition* allows a new program to be formed using the concurrent execution of two existing programs. In Fig. 1, a new program $\mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2$ is created using M_1 and M_2. Note that parallel composition is not necessarily commutative because the outputs of M_1 may be used as inputs to M_2.

Because teleo-reactive programs execute in a truly concurrent manner, we must be able to determine the outputs of a teleo-reactive program.

$$\begin{array}{rcl} out.(O\colon [\![r, g]\!]) & \widehat{=} & O \\ out.\langle\rangle & \widehat{=} & \{\} \\ out.(\langle c \to \mathsf{M}\rangle \frown S) & \widehat{=} & out.\mathsf{M} \cup out.S \\ out.(\mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2) & \widehat{=} & out.\mathsf{M}_1 \cup out.\mathsf{M}_2 \end{array}$$

To ensure that the programs we specify are implementable, we define a number of healthiness constraints on the program. The behaviour of any action $O\colon [\![r, g]\!]$ may not assume properties of the outputs. Hence we require:

$$r \in IntvPred_V \text{ for some } V \subseteq Var \backslash O \qquad \text{for any action } O\colon [\![r, g]\!] \tag{12}$$

For a guarded sequence of programs, we disallow Zeno-like behaviour of the guards. Hence we require:

$$c \text{ is a non-Zeno state predicate for any program } \langle c \to \mathsf{M}\rangle \frown S \tag{13}$$

Finally, two programs executing in parallel may not modify the same outputs. Hence, we require:

$$out.\mathsf{M}_1 \cap out.\mathsf{M}_2 = \{\} \qquad \text{for any program } \mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2 \tag{14}$$

3.2 Semantics

The behaviour of a teleo-reactive program is given by the behaviour function $beh\colon P \to IntvPred$, which is defined in terms of function $beh_F\colon P \to IntvPred$ where F is a set of variables. We assume that $F \supseteq out.\mathsf{M}$ when we write $beh_F.\mathsf{M}$.

Definition 7 *If* M *is a teleo-reactive program and* $F \subseteq Var$ *is a set of variables, then:*

$$
\begin{aligned}
beh_F.(O\colon [\![r, g]\!]) &\;\widehat{=}\; r \Rightarrow g \wedge st.(F \backslash O) && (15)\\
beh_F.\langle\rangle &\;\widehat{=}\; true && (16)\\
beh_F.T &\;\widehat{=}\; ((\boxast c \wedge beh_F.\mathsf{M}) : (\overleftarrow{\neg c} \wedge beh_F.T)) \vee && (17)\\
& \qquad ((\boxast \neg c \wedge beh_F.S) : (\overleftarrow{c} \wedge beh_F.T)) &&\\
beh_F.(\mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2) &\;\widehat{=}\; beh_{F \backslash out.\mathsf{M}_2}.\mathsf{M}_1 \wedge beh_{F \backslash out.\mathsf{M}_1}.\mathsf{M}_2 && (18)
\end{aligned}
$$

By (15), the behaviour of an action a, i.e., $beh_F.a$ states that the guarantee condition g holds and all output variables in F that are not in O are stable provided that the rely condition r holds. The behaviour of an empty sequence of programs, (16), is chaotic, i.e., any behaviour is allowed. By (17), the behaviour of a non-empty sequence of guarded programs, T, is defined recursively — there are two disjuncts corresponding to either $\boxast c$ or $\boxast \neg c$ holding initially on the interval. If $\boxast c$ holds initially, either $\boxast c \wedge beh_F.\mathsf{M}$ holds for the whole interval or the interval may be split into an initial interval in which $\boxast c \wedge beh_F.\mathsf{M}$ holds, followed by an interval in which $\neg c$ holds initially and $beh_F.T$ holds (recursively) for the second interval. Note that each chopped interval must be a maximal interval over which either $\boxast c$ or $\boxast \neg c$ holds. Note that by (13), $beh_F.T$ does not display Zeno-like behaviour, i.e., we cannot split a given finite interval into an infinite partition of finite intervals. By (18), the behaviour of the parallel composition of two programs is defined to be the conjunction of both behaviours, however, we must remove the outputs of M_2 from the when defining the behaviour of M_1 and vice versa.

In a sequence of guarded programs, programs that appear earlier in the sequence are given priority over later programs. For example, in a sequence $\langle c_1 \to \mathsf{M}_1, c_2 \to \mathsf{M}_2\rangle$, if the guard c_1 ever becomes true, then M_2 stops and M_1 begins executing. Hence, the guard of M_2 is effectively $\neg c_1 \wedge c_2$. If neither c_1 nor c_2 holds, then neither M_1 nor M_2 is executed, then any behaviour is allowed [10]. By definition, the variables $out.\mathsf{M}_1 \backslash out.\mathsf{M}_2$ are guaranteed to be stable during execution of M_1 and similarly, variables $out.\mathsf{M}_2 \backslash out.\mathsf{M}_1$ are guaranteed to be stable during execution of M_1.

The next lemma states that a sequence of guarded programs may be decomposed provided $\boxast c$ or $\boxast \neg c$ holds over the given interval.

Lemma 4 *Suppose* S_1*,* S_2 *and* $T \;\widehat{=}\; S_1 \frown \langle c \to \mathsf{M}\rangle \frown S_2$ *are sequences of guarded programs;* $F \subseteq Var$ *is a set of variables; and* r *and* g *are interval predicates. Then:*

$$
\begin{aligned}
\boxast c &\;\Rrightarrow\; (beh_F.T = beh_F.\mathsf{M}) && (19)\\
\boxast \neg c &\;\Rrightarrow\; (beh_F.T = beh_F.(S_1 \frown S_2)) && (20)
\end{aligned}
$$

4 Rely/guarantee

Teleo-reactive programs are reactive, i.e., execute over a dynamic environment, and hence, we use rely/guarantee style reasoning to take the behaviour of the environment into account when reasoning about a program [12]. Here the *rely* condition describes properties of the inputs of the program and the *guarantee* condition describes how the program will behave under the assumption that the rely condition holds.

A teleo-reactive program may not depend on the values of its own output, and hence, we require that the rely condition of a program may only refer to its input variables, however, the guarantee may be a relationship between inputs and outputs.

Definition 8 *Suppose* M *is a teleo-reactive program;* r *and* g *are interval predicates such that* $vars.r \cap out.\mathsf{M} = \{\}$*; and* $F \supseteq out.\mathsf{M}$ *is a set of variables. We define:*

$$F\colon \{r\}\, \mathsf{M}\, \{g\} \quad \widehat{=} \quad r \wedge beh_F.\mathsf{M} \Rrightarrow g$$

Theorem 5 $F\colon \{r\}\, O\colon [\![rr, gg]\!]\, \{g\}$ *holds if* $r \Rrightarrow rr$ *and* $gg \Rrightarrow g$ *hold,* $F \supseteq O$ *and* $vars.r \cap O = \{\}$*.*

We may use the following theorem to prove a property of a sequence of guarded programs.

Theorem 6 *If S and $T \mathrel{\widehat{=}} \langle c \to \mathsf{M}\rangle \frown S$ are sequences of guarded programs; r and g are interval predicates that split and join, respectively; $F \supseteq out.T$; and $vars.r \cap F = \{\}$, then $F\colon \{r\}\, T\, \{g\}$ holds provided that both of the following hold:*

$$F\colon \ \{r\} \quad \mathsf{M} \quad \{\boxast c \Rightarrow g\} \tag{21}$$

$$F\colon \ \{r\} \quad S \quad \{\boxast \neg c \Rightarrow g\} \tag{22}$$

Lemma 7 *Given that S_1 and S_2 are sequences of guarded programs, then $F\colon \{r\}\ S_1 \frown \langle c \to \mathsf{M}\rangle \frown S_2\{\boxast\neg c \Rightarrow g\}$ holds iff $F\colon \{r\}S_1 \frown S_2\{\boxast\neg c \Rightarrow g\}$ holds.*

In program $\mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2$, the behaviours of M_1 and M_2 could conflict if M_1 and M_2 control the same variable. This is especially problematic because we assume true concurrency, as opposed to an interleaved or synchronous execution. One way to resolve conflicts under parallel composition is to split the shared output and derive the final value of the shared output of $\mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2$ (cf [16]). For example, consider a pump (that removes water from a tank) operating in parallel with a hose (that adds water to the tank). Suppose *water_lvl_rate* returns the rate of change of the water level in the tank. Clearly, the pump and hose cannot modify *water_lvl_rate* simultaneously because the pump makes *water_lvl_rate* negative while the hose makes the *water_lvl_rate* positive. To resolve this, we may define *water_in_rate* (only modified by the hose) and *water_out_rate* (only modified by the pump) be the rates at which water is added and removed from the tank, respectively. We may then define $water\_lvl\_rate \mathrel{\widehat{=}} water\_in\_rate - water\_out\_rate$.

Theorem 8 *If $\mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2$ is a teleo-reactive program, $F \supseteq out.(\mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2)$ and $vars.r_1 \cap out.\mathsf{M}_1 = vars.(r_2 \wedge g_1) \cap out.\mathsf{M}_2 = \{\}$ then $F\colon \{r_1 \wedge r_2\}\, \mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2\, \{g_1 \wedge g_2\}$ holds provided both of the following hold:*

$$F\backslash out.\mathsf{M}_2\colon \{r_1\} \quad \mathsf{M}_1 \quad \{g_1\} \tag{23}$$

$$F\backslash out.\mathsf{M}_1\colon \{r_2 \wedge g_1\} \quad \mathsf{M}_2 \quad \{g_2\} \tag{24}$$

Proof 2 *Because $\mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2$ is a teleo-reactive program, $(in.\mathsf{M}_1 \cup out.\mathsf{M}_1) \cap out.\mathsf{M}_2 = \{\}$ holds and we have the following calculation:*

$$\begin{array}{ll}
 & (23) \wedge (24) \\
= & \quad \textit{definition and logic} \\
 & (r_1 \wedge beh_{F\backslash out.\mathsf{M}_2}.\mathsf{M}_1 \Rrightarrow g_1) \wedge (r_2 \wedge beh_{F\backslash out.\mathsf{M}_1}.\mathsf{M}_2 \Rrightarrow (g_1 \Rightarrow g_2)) \\
\Rrightarrow & \quad \textit{logic, weaken antecedents} \\
 & r_1 \wedge r_2 \wedge beh_{F\backslash out.\mathsf{M}_2}.\mathsf{M}_1 \wedge beh_{F\backslash out.\mathsf{M}_1}.\mathsf{M}_2 \Rrightarrow g_1 \wedge (g_1 \Rightarrow g_2) \\
= & \quad \textit{(18), definitions and logic} \\
 & F\colon \{r_1 \wedge r_2\}\, \mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2\, \{g_1 \wedge g_2\}
\end{array}$$

$\square$

Lemma 9 *$F\colon \{r_1 \wedge r_2\}\, \mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2\, \{g_1 \wedge g_2\}$ holds provided both of the following hold:*

$$F\backslash out.\mathsf{M}_2\colon \{r_1\} \quad \mathsf{M}_1 \quad \{g_1\} \tag{25}$$

$$F\backslash out.\mathsf{M}_1\colon \{r_2\} \quad \mathsf{M}_2 \quad \{g_1 \Rightarrow g_2\} \tag{26}$$

The next lemma allows us to prove *simple parallelism* (see Fig. 2), i.e., when the output of M_1 is not used as an input to M_2 and vice versa. We let $\mathsf{M}_1 \| \mathsf{M}_2$ denote the simple parallel composition between M_1 and M_2. Unlike $\overrightarrow{\|}$, programs under simple parallelism are commutative, i.e., $beh_F.(\mathsf{M}_1 \| \mathsf{M}_2) = beh_F.(\mathsf{M}_2 \| \mathsf{M}_1)$.

Lemma 10 (Simple Parallelism) *If $vars.r_1 \cap out.\mathsf{M}_2 = vars.r_2 \cap out.\mathsf{M}_1 = \{\}$ and $F \supseteq out.\mathsf{M}_1 \cup out.\mathsf{M}_2$, then*

$$F\colon \{r_1 \wedge r_2\}\, \mathsf{M}_1 \| \mathsf{M}_2\, \{g_1 \wedge g_2\}$$

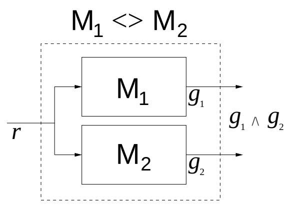

Figure 2: Simple parallelism

holds provided that both of the following hold:

$$F\backslash out.\mathsf{M}_2\colon \{r_1\} \quad \mathsf{M}_1 \quad \{g_1\} \tag{27}$$
$$F\backslash out.\mathsf{M}_1\colon \{r_2\} \quad \mathsf{M}_2 \quad \{g_2\} \tag{28}$$

5 Example

Our example is adapted from the production cell case study [15]. We choose to simplify the problem down to just two programs: a table and a robot arm (see Fig. 3), which is enough to demonstrate our proof technique. A table takes disks from a feed belt and must lower them to the level of the robot, while the robot must fetch disks from the table and deliver them to a depot. We assume an arbitrary number of disks may be placed in the depot.

The controllers for the table and robot are implemented using teleo-reactive programs (see Fig. 5) which we compose in parallel, thus allowing the table and robot to execute independently of each other. Note that we could have implemented the robot grippers as separate program, which would have allowed the robot to rotate while simultaneously opening and closing the grippers. However, for simplicity, we have chosen to allow the grippers to be controlled by the robot program (using actions *Grip* and *Ungrip* in Fig. 5) which allows the robot to rotate or the grippers to open/close, but not together.

5.1 Actions

Movement of the table (T), robot (R) and gripper (G) is controlled by the actions defined in (29) - (34) below. The operating speed of a component C is given by function $\phi.C$. For simplicity, we assume that the acceleration to and deceleration from the operating speed is instantaneous. The program modifies $T.lvl$ (scalar for the height of the

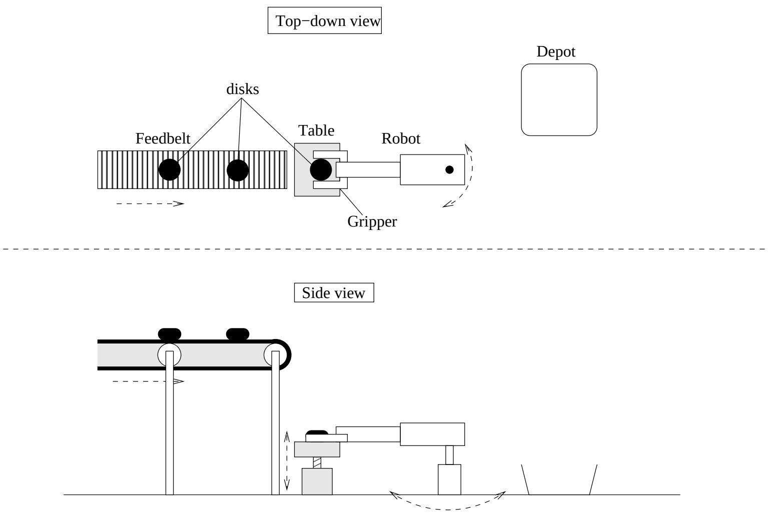

Figure 3: The production cell

table), $G.dist$ (scalar for the distance between grippers) and $R.rot$ (vector for angle of rotation of the robot). We assume max_T and min_T represent the maximum and minimum heights of the table, respectively; that max_G represents the maximum distance between the grippers; and tab, mid and dep are values of $R.rot$ that ensure the robot is rotated towards the table, at a mid-point away from the table and at the depot, respectively.

$$
\begin{aligned}
Nil &\mathrel{\widehat{=}} \{\}\colon [\![true, true]\!] && (29)\\
Raise &\mathrel{\widehat{=}} \{T.lvl\}\colon \left[\!\left[true, \circledast\left(\tfrac{d\,T.lvl}{d\,t} = (\mathsf{if} T.lvl < max\_T \;\mathsf{then}\; \phi.T \;\mathsf{else}\; 0)\right)\right]\!\right] && (30)\\
Lower &\mathrel{\widehat{=}} \{T.lvl\}\colon \left[\!\left[true, \circledast\left(\tfrac{d\,T.lvl}{d\,t} = (\mathsf{if} T.lvl > min\_T \;\mathsf{then}\; -\phi.T \;\mathsf{else}\; 0)\right)\right]\!\right] && (31)\\
Grip &\mathrel{\widehat{=}} \{G.dist\} \left[\!\left[true, \circledast\left(\tfrac{d\,G.dist}{d\,t} = (\mathsf{if} G.dist > 0 \;\mathsf{then}\; -\phi.G \;\mathsf{else}\; 0)\right)\right]\!\right] && (32)\\
Ungrip &\mathrel{\widehat{=}} \{G.dist\}\colon \left[\!\left[true, \circledast\left(\tfrac{d\,G.dist}{d\,t} = (\mathsf{if} G.dist < max\_G \;\mathsf{then}\; \phi.G \;\mathsf{else}\; 0)\right]\!\right]\right. && (33)\\
Rot_{loc} &\mathrel{\widehat{=}} \{R.rot\}\colon \left[\!\!\left[true, \circledast\left(\frac{d\,R.rot}{d\,t} = \begin{pmatrix} \mathsf{if} R.rot = loc \;\mathsf{then}\; 0 \\ \mathsf{elseif} R.rot < loc \;\mathsf{then}\; \phi.R \\ \mathsf{else} - \phi.R \end{pmatrix} \right) \right]\!\!\right] && (34)
\end{aligned}
$$

By (29), Nil has no inputs or outputs and hence does nothing. By (30), the $Raise$ action modifies $T.lvl$ and guarantees that the rate of change of $T.lvl$ is $\phi.T$ at each point of the given interval. Conditions (31) - (34) are similar.

5.2 Program

The program uses constants FB_lvl and R_lvl (scalars for the height of the feed belt and robot, respectively), dw (scalar for width of a disk), R_arm_len (scalar for the robot arm length) and R_pos (vector for the position of the robot). Arithmetic operations on vectors are assumed to be defined in the normal manner. We assume $Disk$ represents the set of all disks in the system and for each $disk \in Disk$, we use $disk.pos$ (vector for the current position of the center of $disk$) and $disk.lvl$ (scalar for the current height of $disk$) to determine the position of $disk$. We define $G.pos$ (vector for the gripper position) using the robot position, the length of the robot arm, the width of the disk and the robot rotation as follows:

$$G.pos \mathrel{\widehat{=}} R\_pos + (R\_arm\_len + \tfrac{dw}{2}, R.rot)$$

the following predicates are used to determine specific positions of $disk$ in the system, where constants T_pos and D_pos are vectors for the position of the table and depot, respectively.

$$
\begin{aligned}
onT.disk &\mathrel{\widehat{=}} (disk.pos = T\_pos) \wedge (disk.lvl = T.lvl)\\
atG.disk &\mathrel{\widehat{=}} (disk.pos = G.pos) \wedge (disk.lvl = R\_lvl)\\
inD.disk &\mathrel{\widehat{=}} (disk.pos = D\_pos) \wedge (disk.lvl = 0)\\
hbR.disk &\mathrel{\widehat{=}} atG.disk \wedge (G.dist = dw)
\end{aligned}
$$

Predicates $onT.disk$, $atG.disk$ and $onR.disk$ hold if $disk$ is on the table, at the gripper location and being held by the grippers, respectively. To detect possible collisions between the table and the robot arm we define a set of vectors T_area corresponding to a set of $G.pos$ values for which the table and robot arm collide. We note that the table and robot arm may overlap even if $G.pos \neq T\_pos$ holds.

We define a number of predicates which serve as shorthand for determining the positions of the various components. These predicates are implemented as sensors in the production cell.

$$
\begin{aligned}
T\_at\_FB &\mathrel{\widehat{=}} T.lvl = FB\_lvl & \qquad G\_at\_T &\mathrel{\widehat{=}} G.pos = T\_pos\\
T\_at\_R &\mathrel{\widehat{=}} T.lvl = R\_lvl & G\_at\_D &\mathrel{\widehat{=}} G.pos = D\_pos\\
full &\mathrel{\widehat{=}} \exists disk\colon Disk \bullet onT.disk & G\_open &\mathrel{\widehat{=}} G.dist = max\_G\\
holding &\mathrel{\widehat{=}} \exists disk\colon Disk \bullet hbR.disk & G\_near\_T &\mathrel{\widehat{=}} G.pos \in T\_area
\end{aligned}
$$

Thus, T_at_FB holds iff the level of the table is equal to the constant FB_lvl. The other predicates are similar. The teleo-reactive programs for controlling the table and robot of the production cell are provided in Figures 4 and 5, respectively.

The table only operates (i.e., executes runT) over an interval in which $\neg GnearT$ holds. Thus, the table does not move while the robot arm is in the way. The program runT lowers the table by executing action $Lower$ while

$$\mathsf{Robot} \mathrel{\widehat{=}} \left\langle \begin{array}{r} holding \rightarrow \mathsf{drop\_at\_depot}, \\ full \wedge T\_at\_R \rightarrow \mathsf{pickup}, \\ true \rightarrow Rot_{mid} \end{array} \right\rangle$$

$$\mathsf{Table} \mathrel{\widehat{=}} \left\langle \begin{array}{r} \neg GnearT \rightarrow \mathsf{runT}, \\ true \rightarrow Nil \end{array} \right\rangle$$

$$\mathsf{drop\_at\_depot} \mathrel{\widehat{=}} \left\langle \begin{array}{r} G\_at\_D \rightarrow Ungrip, \\ true \rightarrow Rot_{dep} \end{array} \right\rangle$$

$$\mathsf{runT} \mathrel{\widehat{=}} \left\langle \begin{array}{r} full \wedge \neg T\_at\_R \rightarrow Lower, \\ \neg full \wedge \neg T\_at\_FB \rightarrow Raise, \\ true \rightarrow Nil \end{array} \right\rangle$$

$$\mathsf{pickup} \mathrel{\widehat{=}} \left\langle \begin{array}{r} G\_at\_T \wedge G\_open \rightarrow Grip, \\ G\_open \rightarrow Rot_{tab}, \\ true \rightarrow Ungrip \end{array} \right\rangle$$

Figure 4: Table controller

Figure 5: Robot controller

it is full and not yet at the robot level. Execution of runT raises the table by executing *Raise* while $\neg(full \wedge \neg T\_at\_R) \wedge (\neg full \wedge \neg T\_at\_FB)$ holds, which simplifies to $\neg full \wedge \neg T\_at\_FB$. The table executes the *Nil* action (which does nothing) over an interval in which the guards of *Lower* and *Raise* are false. Note that in the context of the Table program, each of the guards of runT has $\neg GnearT$ as an additional conjunct.

While it is holding a disk, the Robot program executes drop_at_depot, which places the disk it is holding in the depot. Robot executes pickup while it is not holding a disk, the table is full and is at the robot level, which picks up a disk from the table. While there is no disk to be picked up or dropped off, Robot executes Rot_{mid}, which moves the gripper away from the table. Program drop_at_depot executes *Ungrip* while the gripper is already at the depot, otherwise, it rotates towards the depot. Program pickup executes *Grip* while the grippers are at the table and the distance between the grippers exceeds the width of a disk. While the grippers are not at the table, but the grippers are open far enough, pickup rotates the robot to the table. The default action of pickup is to open the grippers by executing *Ungrip*.

The overall system is constructed using simple parallelism as follows:

$$TR \quad \widehat{=} \quad \mathsf{Table} \parallel \mathsf{Robot}$$

Although the component programs themselves are simple, *TR* allows the programs in Figures 4 and 5 to execute in true parallelism to perform the complex task of transporting a disk from the feed belt to the depot.

5.3 A safety proof

A safety requirement of the system is that the robot does not collide with the other components. Using the configuration of the system, we can rule out collisions between the robot and the depot, but it may be possible for the robot to collide with the table. Thus, we obtain a safety requirement:

$$TR\colon \{true\} \quad \mathsf{TR} \quad \{inv.(GnearT \Rightarrow T\_at\_R)\} \tag{35}$$

Although it is tempting to use Lemma 10 and split the proof into Table and Robot components, a proof using Lemma 10 is not possible because the value of $inv.(GnearT \Rightarrow T\_at\_R)$ is modified by both Table and Robot. Instead, we obtain the following calculation:

$$\begin{array}{ll} & (35) \\ \Leftarrow & \quad \text{logic} \\ & TR\colon \{true\}\, TR\, \{\circledast GnearT \wedge \circledast\neg T\_at\_R \Rightarrow \downarrow(GnearT \wedge \neg T\_at\_R)\} \\ \Leftarrow & \quad \text{Lemma 7} \\ & TR\colon \{true\}\, Nil \left\| \left\langle \begin{array}{l} holding \rightarrow \mathsf{drop\_at\_depot}, \\ true \rightarrow Rot_{mid} \end{array} \right\rangle \right. \{\circledast(GnearT \wedge \neg T\_at\_R) \Rightarrow \downarrow(GnearT \wedge \neg T\_at\_R)\} \end{array}$$

$=$ logic

$$TR\colon \{true\}\ Nil \,\Big\|\, \left\langle \begin{array}{l} holding \to \mathsf{drop\_at\_depot}, \\ true \to Rot_{mid} \end{array} \right\rangle \ \{inv.(GnearT \Rightarrow T\_at\_R)\}$$

$\Leftarrow$ Lemma 9

$$T\colon \{true\}\ Nil\ \{st.(T.lvl)\} \ \wedge$$
$$R\colon \{true\} \left\langle \begin{array}{l} holding \to \mathsf{drop\_at\_depot}, \\ true \to Rot_{mid} \end{array} \right\rangle \{st.(T.lvl) \Rightarrow inv.(GnearT \Rightarrow T\_at\_R)\}$$

$\Leftarrow$ first triple: Theorem 5
second triple: logic, use $st.(T.lvl)$

$$R\colon \{true\} \left\langle \begin{array}{l} holding \to \mathsf{drop\_at\_depot}, \\ true \to Rot_{mid} \end{array} \right\rangle \{inv.(\neg GnearT)\}$$

$\Leftarrow$ Theorem 6 twice

$$R\colon \{true\}\ Ungrip\ \{\boxast holding \wedge \boxast G\_at\_D \Rightarrow inv.(\neg GnearT)\}$$
$$R\colon \{true\}\ Rot_{dep}\ \{\boxast holding \wedge \boxast \neg G\_at\_D \Rightarrow inv.(\neg GnearT)\}$$
$$R\colon \{true\}\ Rot_{mid}\ \{\boxast \neg holding \Rightarrow inv.(\neg GnearT)\}$$

$\Leftarrow$ $\boxast G\_at\_D \Rightarrow \boxast \neg GnearT$, $beh_R.Rot_{dep} \vee beh_R.Rot_{mid} \Rightarrow inv.(\neg GnearT)$

$true$

5.4 A progress proof

A progress requirement of the system is that

> "Any disk on the table is eventually at the depot."

This can be ensured by showing that each disk reaches the next component in the production line. That is, each disk on the table is eventually held by the robot, i.e.,

$$\{r_1 \wedge (\ell \geq \epsilon)\} \quad TR \quad \{\overleftarrow{onT.disk} \Rightarrow \nabla \overleftarrow{hbR.disk}\} \tag{36}$$

and each disk being held by the robot is eventually placed in the depot, i.e.,

$$\{r_2 \wedge (\ell \geq \kappa)\} \quad TR \quad \{\overleftarrow{hbR.disk} \Rightarrow \nabla \overleftarrow{inD.disk}\} \tag{37}$$

We present a detailed proof of (36), and elide the details of (37), which are mostly similar to (36). The proof of (37) is less complicated because it only involves interaction between the robot and the environment, as opposed to the table, robot and environment in the case of (36).

(36)

$\Leftarrow$ Definition 8 and logic

$$\{r_1 \wedge (\ell \geq \epsilon)\}\ TR\ \{\overleftarrow{onT.disk} \wedge \boxast \neg hbR.disk \Rightarrow \nabla \overleftarrow{hbR.disk}\}$$

To prove the above, we assume a property on the movement of the disk. In particular, we require:

$$r_1 \Rrightarrow \forall T.lvl, R.rot, G.dist \bullet \overleftarrow{onT.disk} \wedge \boxast \neg hbR.disk \Rightarrow \boxast onT.disk$$

which states that if the disk is on the table at the start of an interval and is not held by the robot throughout the interval, then the disk remains on the table throughout the interval. Note that none of the free variables of r_1 are outputs of TR. The rely condition r_1 allows us to simplify the guarantee as follows:

$$\{r_1 \wedge (\ell \geq \epsilon)\}\ TR\ \{\boxast onT.disk \Rightarrow \nabla \overleftarrow{hbR.disk}\}$$

The significance of this calculation is that we can now assume that the disk stays on the table, as opposed to being on the table at the start of the interval. Using Lemma 3 (transitivity) and assuming $\epsilon = \epsilon_1 + \epsilon_2$, the condition above holds if we can prove both of the following:

$$\{r_1 \wedge (\ell \geq \epsilon_1)\} \quad TR \quad \{\boxast onT.disk \Rightarrow \nabla \overleftarrow{T\_at\_R}\} \tag{38}$$
$$\{r_1 \wedge (\ell \geq \epsilon_2)\} \quad TR \quad \{\boxast onT.disk \wedge \overleftarrow{T\_at\_R} \Rightarrow \nabla \overleftarrow{hbR.disk}\} \tag{39}$$

Thus, to show that a disk on the table is eventually held by the robot, we must show (38), i.e., that the table eventually reaches the robot level. Furthermore, by (39), if a full table is at the robot level, then the disk must eventually be held by the robot. The proof of (38) uses:

$$\{true\} \quad TR \quad \{inv.(R\_lvl \leq T.lvl \leq FB\_lvl)\} \tag{40}$$

which is an easily provable safety condition.

Proof of (38).

$$\begin{array}{ll}
& \{r_1 \wedge (\ell \geq \epsilon_1)\}\, TR\, \{\boxast onT.disk \Rightarrow \nabla\overleftarrow{T_at_R}\} \\
\Leftarrow & \quad \text{logic, } \boxast(onT.disk \Rightarrow full) \\
& \{r_1 \wedge (\ell \geq \epsilon_1)\}\, TR\, \{\boxast(full \wedge \neg T_at_R) \Rightarrow \uparrow T_at_R\} \\
\Leftarrow & \quad \text{(35), parallel composition (18)} \\
& \{r_1 \wedge (\ell \geq \epsilon_1)\}\, \mathsf{Table}\, \{\boxast(full \wedge \neg T_at_R \wedge \neg GnearT) \Rightarrow \uparrow T_at_R\} \\
\Leftarrow & \quad \text{(19) and (20)} \\
& \{r_1 \wedge (\ell \geq \epsilon_1)\}\, Lower\, \{\boxast(full \wedge \neg T_at_R \wedge \neg GnearT) \Rightarrow \uparrow T_at_R\} \\
\Leftarrow & \quad \text{(31) (i.e., definition of } Lower\text{), (40) and assumption } r_1 \\
& true
\end{array}$$

Proof of (39). This proof uses the following trivially provable properties:

$$\{true\} \quad \mathsf{Table} \quad \{\overleftarrow{T\_at\_R}\ \mathbf{wu}\ \neg\overleftarrow{full}\} \tag{41}$$

which states if the table is at the robot level the table is full, then the table remains at the robot level unless the table is not full. The proof of (41) follows directly from the behaviour of Table. Thus, we obtain:

$$\begin{array}{ll}
& (39) \\
\Leftarrow & \quad \text{using (41)} \\
& \{r_1 \wedge (\ell \geq \epsilon_2)\}\, TR\, \{\boxast(onT.disk \wedge T_at_R) \Rightarrow \nabla\overleftarrow{hbR.disk}\}
\end{array}$$

As before, we can now assume the table remains at the robot level throughout the interval as opposed to only at the start. Assuming $\epsilon_2 = \epsilon_{21} + \epsilon_{22}$, we apply Lemma 3 (transitivity) to obtain the following cases:

$$\{r_1 \wedge (\ell \geq \epsilon_{21})\} \quad TR \quad \{\boxast(onT.disk \wedge T\_at\_R) \Rightarrow \nabla\overleftarrow{\neg holding}\} \tag{42}$$

$$\{r_1 \wedge (\ell \geq \epsilon_{22})\} \quad TR \quad \{\boxast(onT.disk \wedge T\_at\_R) \wedge \overleftarrow{\neg holding} \Rightarrow \nabla\overleftarrow{hbR.disk}\} \tag{43}$$

Thus, by (42) for the robot to hold the disk on the table, the robot must eventually not be holding anything. Furthermore, by (43) if the disk is on the table, the table is at the robot level and the robot is not holding anything, then the robot must eventually hold the disk. The first case, i.e., (42) is proved as part of (37) and hence we elide the details.

Proof of (43). The proof uses the following trivial safety property:

$$\{true\} \quad \mathsf{Robot} \quad \{\boxast full \wedge \overleftarrow{\neg holding} \Rightarrow \boxast\neg holding\} \tag{44}$$

then obtain the following calculation:

$$\begin{array}{ll}
& (43) \\
\Leftarrow & \quad \text{(44) because } onT.disk \Rightarrow full \\
& \{r_1 \wedge (\ell \geq \epsilon_{22})\}\, TR\, \{\boxast(onT.disk \wedge T_at_R \wedge \neg holding) \Rightarrow \nabla\overleftarrow{hbR.disk}\} \\
\Leftarrow & \quad \text{Theorem 8} \\
& \{r_1 \wedge (\ell \geq \epsilon_{22})\}\, \mathsf{Robot}\, \{\boxast(onT.disk \wedge T_at_R \wedge \neg holding) \Rightarrow \nabla\overleftarrow{hbR.disk}\}
\end{array}$$

The rely condition above states that the interval is of length ϵ_{22} or greater and throughout the interval *disk* is on the table, the table is at the robot level and the robot is not holding a disk. The proof that the robot eventually holds *disk* under this rely condition is straightforward because we are only required to consider execution of the Robot program in isolation. For such proofs we may use the techniques described in [7] and hence, the details of the proof are elided.

6 Other composition operators

Besides hierarchical and parallel composition, teleo-reactive programs may also be composed using hiding (Section 6.1), feedback (Section 6.2) and pipelines (Section 6.3), which is derived by combining of parallel composition and hiding.

6.1 Hiding

We define hiding as a basic form of composition that allows variables of a program to be hidden so that they may not be used by any other program, including the environment (see Fig. 6). Hiding is used to derive the pipeline operator. For a program M and a set of variables $m \subseteq out.\mathsf{M}$, we use $\mathsf{M}\backslash m$ to denote a program in which m is hidden from the environment. The outputs of program $\mathsf{M}\backslash m$ is defined as:

$$out.(\mathsf{M}\backslash m) \quad \widehat{=} \quad out.\mathsf{M}\backslash m$$

and define the behaviour of $\mathsf{M}\backslash m$ in a possibly larger frame $F \supseteq out.(\mathsf{M}\backslash m)$ is defined as follows:

$$beh_{F\backslash m}.(\mathsf{M}\backslash m) \quad \widehat{=} \quad \exists m \bullet beh_F.\mathsf{M} \tag{45}$$

The following theorem allows us to prove properties of a program after an output is hidden.

Theorem 11 (Hiding) *If $m \subseteq out.\mathsf{M}$, $F \supseteq out.\mathsf{M}$ and $F\colon \{r\}\,\mathsf{M}\,\{g\}$, then $F\backslash m\colon \{r\}\,\mathsf{M}\backslash m\,\{\exists m \bullet g\}$.*

Proof 3 *Because $m \subseteq out.\mathsf{M}$, the variables in m do not not occur free in r. Hence, we obtain the following calculation:*

$$\begin{array}{ll}
 & F\backslash m\colon \{r\}\,\mathsf{M}\backslash m\,\{\exists m \bullet g\} \\
= & \quad \textit{expand triple, (45)} \\
 & r \wedge (\exists m \bullet beh_F.\mathsf{M}) \Rrightarrow \exists m \bullet g \\
\Leftarrow & \quad m \textit{ nfi } r \\
 & (\exists m \bullet r \wedge beh_F.\mathsf{M}) \Rrightarrow \exists m \bullet g \\
\Leftarrow & \quad \textit{logic} \\
 & F\colon \{r\}\,\mathsf{M}\,\{g\}
\end{array}$$

□

6.2 Feedback

Feedback allows us to use the output of a component as an input to the same component. A natural method of reasoning about feedback is to use fixed points with delay [19, 6]. However, because this approach is potentially complex, we prefer the method of Mahoney et al, where introduction of feedback is viewed as strengthening of the initial specification to require that the output has the same value as the input [13, 6].

Fig. 6 denotes the program where the outputs h are fed back as inputs The outputs of program with feedback include the variables being fed back to the program, i.e.,

$$out.(\mu\, e\backslash h \bullet \mathsf{M}) \quad \widehat{=} \quad out.\mathsf{M} \cup e$$

This means that the rely condition of $\mu\, e\backslash h \bullet \mathsf{M}$ may not refer to input variables h. The behaviour of a program is defined to the original program, but with input variables replaced by their output values. That is:

$$beh_F.(\mu\, e\backslash h \bullet \mathsf{M}) \quad \widehat{=} \quad (beh_F.\mathsf{M})[e\backslash h] \tag{46}$$

The following theorem allows one to prove properties of components with feedback.

Theorem 12 (Feedback) *If $F \supseteq out.\mathsf{M}$, $vars.r \cap out.\mathsf{M} = vars.r_1 \cap out.(\mu\, e\backslash h \bullet out.\mathsf{M}) = \{\}$, $F\colon \{r\}\,\mathsf{M}\,\{g\}$ and $F\colon \{r_1\}\ \mu\, e\backslash h \bullet \mathsf{M}\,\{r[e\backslash h]\}$ then $F\colon \{r_1\}\ \mu\, e\backslash h \bullet \mathsf{M}\,\{g[e\backslash h]\}$.*

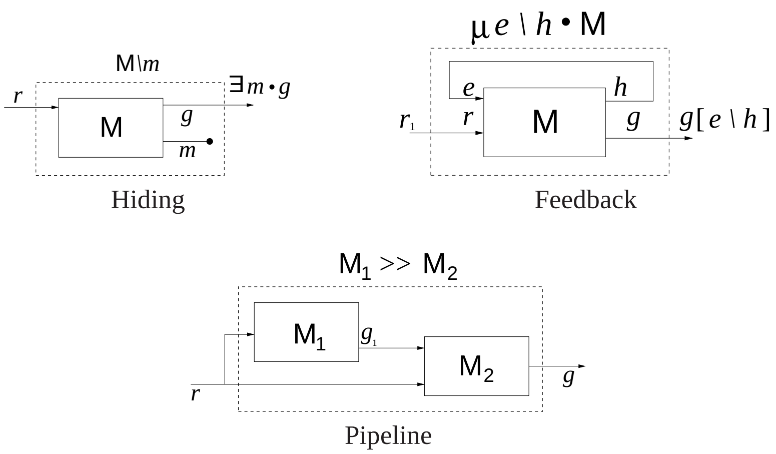

Figure 6: Further composition operators

Proof 4

$$
\begin{array}{ll}
 & F\colon \{r_1\}\ \mu\, e\backslash h \bullet \mathsf{M}\ \{g[e\backslash h]\} \\
= & \quad \textit{definitions} \\
 & r_1 \wedge beh_F.\mathsf{M}[e\backslash h] \Rrightarrow g[e\backslash h] \\
\Leftarrow & \quad \textit{assumption: } r_1 \wedge beh_F.\mathsf{M}[e\backslash h] \Rrightarrow r[e\backslash h] \\
 & r[e\backslash h] \wedge beh_F.\mathsf{M}[e\backslash h] \Rrightarrow g[e\backslash h] \\
= & \quad \textit{logic} \\
 & (r \wedge beh_F.\mathsf{M} \Rrightarrow g)[e\backslash h] \\
\Leftarrow & \quad \textit{assumption: } F\colon \{r\}\ \mathsf{M}\ \{g\} \\
 & \textit{true}
\end{array}
$$

□

In addition to the program with no feedback establishing g under rely condition r, the theorem requires that the program extended with feedback reestablish r with fed back inputs e replaced by outputs h.

The lemma below states that replacing a component M by a component $\mathsf{M}' \mathrel{\widehat{=}} \mu\, e\backslash h \bullet \mathsf{M}$ within a guarded program $T \mathrel{\widehat{=}} \langle c \to \mathsf{M}\rangle \frown S$, then the behaviour of $\mu\, e\ h \bullet T$ is equivalent to the program $\mu\, e\backslash h \bullet \langle c \to \mathsf{M}'\rangle \frown S$.

Lemma 13 *If $T \mathrel{\widehat{=}} \langle c \to \mathsf{M}\rangle \frown S$, $T' \mathrel{\widehat{=}} \langle c \to \mu\, e\backslash h \bullet \mathsf{M}\rangle \frown S$ and $F \supseteq out.T$ then*

$$beh_F.(\mu\, e\backslash h \bullet T') \quad \equiv \quad beh_F.(\mu\, e\backslash h \bullet T)$$

Proof 5

$$
\begin{array}{ll}
 & beh_F.(\mu\, e\backslash h \bullet T').\Delta \\
= & \quad \textit{definition of feedback} \\
 & beh_F.(T'[e\backslash h]).\Delta \\
= & \quad \textit{logic} \\
 & \exists \delta\colon NZpart.\Delta \bullet \forall i\colon \mathrm{dom}\,.\delta \bullet ((\Box c \wedge beh_F.(\mu\, e\backslash h \bullet \mathsf{M}))[e\backslash h]).\delta_i \vee ((\Box\neg c \wedge beh_F.S)[e\backslash h]).\delta_i \\
= & \quad \textit{definition of feedback, logic} \\
 & \exists \delta\colon NZpart.\Delta \bullet \forall i\colon \mathrm{dom}\,.\delta \bullet ((\Box c \wedge beh_F.\mathsf{M})[e\backslash h]).\delta_i \vee ((\Box\neg c \wedge beh_F.S)[e\backslash h]).\delta_i \\
= & \quad \textit{beh definition} \\
 & ((beh_F.T)[e\backslash h]).\Delta \\
= & \quad \textit{beh definition} \\
 & beh_F.(\mu\, e\backslash h \bullet T).\Delta
\end{array}
$$

We provide a concrete example by considering an oscillator that is constructed using an inverter, inv and a feedback loop. We let booleans on_e and on be the input and output of inv, respectively. We assume that on is initially *false*, and that inv inverts the value of on_e after a delay of length d. More formally, the behaviour of inv is defined by:

$$beh_F.\mathsf{inv} \mathrel{\widehat{=}} \quad \forall t\colon Time \bullet (t < \epsilon \Rightarrow \neg on@t) \wedge (on@(t+\epsilon) = \neg on_e@t)$$

Now, given the following rely condition:

$$rely.\Delta \mathrel{\widehat{=}} \exists \delta: NZpart.\Delta \bullet (\forall i: \mathrm{dom}\, .\delta \bullet \ell.\delta_i = \epsilon) \wedge alt.on_e.\delta \wedge \neg on_e.\delta_0$$

which states that the value of on_e flips after every ϵ time units, we have

$$F: \{rely\}\ \mathsf{inv}\ \{rely[on_e \backslash on]\} \tag{47}$$

That is, given that the value of input on_e oscillates every ϵ units, the inverter is guaranteed to oscillate the value of output on. The oscillator osc uses inv and feeds the output on back to the input on_e. That is, we define

$$\mathsf{osc} \mathrel{\widehat{=}} \mu\, on_e \backslash on \bullet \mathsf{inv}.$$

We prove our desired property of the oscillator:

$$F: \{true\}\ \mathsf{osc}\ \{rely[on_e \backslash on]\}$$

using Theorem 12, (47) and the trivial property $F: \{rely\}\ \mu\, on_e \backslash on \bullet \mathsf{inv}\{rely[on_e \backslash on]\}$.

Although development of systems with feedback is necessary for reasoning at an absolute level of precision, we aim to incorporate the time bands logic [5] into the teleo-reactive framework. Thus, issues that require feed back at an absolute level of precision (e.g., a program does not modify its own input) are absent in the context of time bands.

6.3 Pipelines

A *pipeline* is a special case of parallel composition where all outputs of one first component become inputs to another and the outputs of the first component are hidden from the environment of the pipeline. We use $\mathsf{M}_1 \gg \mathsf{M}_2$ to denote the pipeline from M_1 to M_2 (see Fig. 6), which is defined as follows:

$$\mathsf{M}_1 \gg \mathsf{M}_2 \mathrel{\widehat{=}} (\mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2) \backslash out.\mathsf{M}_1 \tag{48}$$

hence, we have

$$out.(\mathsf{M}_1 \gg \mathsf{M}_2) = out.\mathsf{M}_2$$

Pipelines inherit the healthiness conditions of parallel composition, and hence, their behaviour in a context C is only defined if the healthiness conditions of the parallel composition hold.

Lemma 14 (Pipeline) *If* $out.\mathsf{M}_1 \cap vars.(r_1 \wedge r_2) = out.\mathsf{M}_1 \cap g = \{\}$, *then*

$$F \backslash out.\mathsf{M}_1: \{r_1 \wedge r_2\}\ \mathsf{M}_1 \gg \mathsf{M}_2\ \{g\}$$

holds provided that both of the following hold:

$$F \backslash out.\mathsf{M}_2: \{r_1\} \quad \mathsf{M}_1 \quad \{g_1\} \tag{49}$$

$$F \backslash out.\mathsf{M}_1: \{r_2 \wedge g_1\} \quad \mathsf{M}_2 \quad \{g\} \tag{50}$$

Proof 6

$$\begin{array}{ll}
 & F \backslash out.\mathsf{M}_1: \{r_1 \wedge r_2\}\ \mathsf{M}_1 \gg \mathsf{M}_2\ \{g\} \\
= & \quad \textit{(48) and definitions} \\
 & F \backslash out.\mathsf{M}_1: \{r_1 \wedge r_2\}\ (\mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2) \backslash out.\mathsf{M}_1\ \{g\} \\
= & \quad \textit{Theorem 11, } out.\mathsf{M}_1 \textit{ does not occur free in } r_1 \wedge r_2 \textit{ and } g \\
 & F: \{r_1 \wedge r_2\}\ \mathsf{M}_1 \overrightarrow{\|} \mathsf{M}_2\ \{g\} \\
\Leftarrow & \quad \textit{Theorem 8 with } g_2 \textit{ replaced by } g \\
 & (49) \wedge (50)
\end{array}$$

□

7 Conclusion

Teleo-reactive programs present a novel high-level approach to programming and differ considerably from other real-time frameworks. A formal framework for reasoning about teleo-reactive programs has thus far not been developed. The semantics of a single process teleo-reactive program are provided in [7, 10]. This paper revises this logic and provides techniques for reasoning about teleo-reactive programs under various composition operators: renaming, hiding, and parallel composition (including special cases pipelines and simple parallelism).

We note that the logic developed in this paper does not yet cover all the nuances of real-time systems. In particular, we have assumed perfect sampling, i.e., that all sensors are sampled simultaneously, and hence each sampled state corresponds to a real state of the system. However, in a real system, sensors are usually sampled one at a time, and hence, these systems can suffer from sampling errors [5]. We plan to encode a sampling logic into this theory as part of future work.

Acknowledgements This research is supported by Australian Research Council (ARC) Discovery Grant, *Combining Time Bands and Teleo-Reactive Programs for Advanced Dependable Real-Time Systems* (DP0987452) and The University of Queensland New Staff Research Fund.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] R. Alur and T. A. Henzinger. Reactive modules. *Form. Methods Syst. Des.*, 15(1):7–48, 1999.

[3] R-J. R. Back, L. Petre, and I. Porres. Generalizing action systems to hybrid systems. In Mathai Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Computer Science*, pages 73–91. Springer Berlin / Heidelberg, 2000.

[4] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. J. Pappas. Symbolic planning and control of robot motion: State of the art and grand challenges. *IEEE Robotics and Automation Magazine*, 14(1):61–70, March 2007.

[5] A. Burns and I. J. Hayes. A timeband framework for modelling real-time systems. *Real-Time Systems*, 45(1):106–142, 2010.

[6] T. Cant, B. P. Mahony, and J. McCarthy. *Design oriented verification and evaluation : The DOVE project*. DSTO Information Sciences Laboratory, Edinburgh, S. Aust., 2002.

[7] B. Dongol, I. J. Hayes, and P. J. Robinson. Reasoning about real-time teleo-reactive programs. Technical Report SSE-2010-01, School of ITEE, The University of Queensland, Australia, 2010.

[8] C. A. Furia, M. Rossi, D. Mandrioli, and A. Morzenti. Automated compositional proofs for real-time systems. *Theor. Comput. Sci.*, 376(3):164–184, 2007.

[9] A. Gargantini and A. Morzenti. Automated deductive requirements analysis of critical systems. *ACM Trans. Softw. Eng. Methodol.*, 10:255–307, July 2001.

[10] 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.

[11] J. Hooman. Compositional verification of real-time applications. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *COMPOS*, volume 1536 of *Lecture Notes in Computer Science*, pages 276–300. Springer, 1997.

[12] C. B. Jones. Tentative steps toward a development method for interfering programs. *AMC Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[13] P. Katis, N. Sabadini, and R. F. C. Walters. Span(graph): A categorial algebra of transition systems. In Michael Johnson, editor, *AMAST*, volume 1349 of *LNCS*, pages 307–321. Springer, 1997.

[14] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[15] C. Lewerentz and T. Lindner, editors. *Formal Development of Reactive Systems - Case Study Production Cell*, volume 891 of *LNCS*. Springer, 1995.

[16] B. P. Mahony, C. Millerchip, and I. J. Hayes. A boiler control system: A case study in timed refinement. In Diana Del Bel Belluz, editor, *Technical proceedings International Symposium on Design and Review of Software-Controlled Safety-Related Systems, Ottawa*, June 1993. 50 pages.

[17] Z. Manna and A. Pnueli. *Temporal Verification of Reactive and Concurrent Systems: Specification*. Springer-Verlag New York, Inc., 1992.

[18] L. Meinicke and I. J. Hayes. Continuous action system refinement. In T. Uustalu, editor, *MPC*, volume 4014 of *LNCS*, pages 316–337. Springer, 2006.

[19] O. Müller and P. Scholz. Functional specification of real-time and hybrid systems. In Oded Maler, editor, *HART*, volume 1201 of *LNCS*, pages 273–285. Springer, 1997.

[20] Nils J. Nilsson. Teleo-reactive programs and the triple-tower architecture. *Electronic Transactions on Artificial Intelligence*, 5:99–110, 2001.

[21] M. Rönkkö, A. P. Ravn, and K. Sere. Hybrid action systems. *Theor. Comput. Sci.*, 290(1):937–973, 2003.

[22] C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. EATCS: Monographs in Theoretical Computer Science. Springer, 2004.

[23] P. Zhou, J. Hooman, and R. Kuiper. Compositional verification of real-time systems with explicit clock temporal logic. *Formal Asp. Comput.*, 8(3):294–323, 1996.