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.

\[
\begin{align*}
\text{Lift} & \equiv \begin{cases} 
\text{door\_closed} \rightarrow \text{runLift}, \\
\text{true} \rightarrow \text{Nil}
\end{cases} \\
\text{runLift} & \equiv \begin{cases} 
\text{lift\_full} \land \neg \text{bottom} \rightarrow \text{Lower}, \\
\text{lift\_empty} \land \neg \text{top} \rightarrow \text{Raise}, \\
\text{true} \rightarrow \text{Nil}
\end{cases}
\end{align*}
\]

The main program Lift executes program runLift in any interval in which the door is closed, i.e., door\_closed holds and executes Nil (which does nothing) otherwise. Program 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 runLift is executed while door_closed holds and Nil (which does nothing) is executed otherwise. If door_closed ever becomes false while runLift is executing, then runLift stops and Nil starts executing. Thus, Lift is equivalent to (door_closed → runLift, ¬door_closed → Nil). Teleo-reactive programs also naturally support hierarchical composition, e.g., the runLift program executes within the context of the door_closed guard, i.e., each guard in 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.

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 $\text{Time} \rightarrow \text{Val}$ (where $\text{Time} \approx \mathbb{R}$), 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

\[
\text{Interval} \equiv \{ \Delta \subseteq \mathbb{R} \mid \Delta \neq \emptyset \land \forall t, t' \in \Delta \cdot t < t' \Rightarrow \forall t'' : \mathbb{R} \cdot t < t'' \Rightarrow t'' \in \Delta \}
\]

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

\[
\Delta \propto \Delta' \equiv (\text{lub} \Delta = \text{glb} \Delta') \land (\Delta \cup \Delta' \in \text{Interval}) \land (\Delta \cap \Delta = \{\})
\]

That is, \( \Delta \propto \Delta' \) states that \( \Delta' \) is an interval that immediately follows \( \Delta \).

We define a state space as \( \Sigma \in V \rightarrow \text{Val} \) where \( V \subseteq \text{Var} \) is a set of variables and \( \text{Val} \) 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 \( \mathcal{P}X \in X \rightarrow \mathbb{B} \), a state is a member of \( \Sigma \), and a state predicate is a member of \( \mathcal{P}\Sigma \). The (real-time) stream is given by \( \text{Stream}_V \in \mathcal{P}X \rightarrow \Sigma \), which is a total function from times to states with variables \( V \). A stream predicate is a member of \( \mathcal{P}\text{Stream}_V \) and an interval predicate is a member of the set \( \text{IntvPred}_V \equiv \text{Interval} \rightarrow \mathcal{P}\text{Stream}_V \). Interval predicates allow us to reason about the behaviour of a stream with respect to a given interval. We let \( \text{vars.c} \) and \( \text{vars.p} \) denote the sets of all variables \( V \) that may occur free in \( c \in \mathcal{P}\Sigma \) and \( p \in \text{IntvPred}_V \).

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

\[
\begin{align*}
(sp_1 \Rightarrow sp_2) & \equiv \forall \text{tr}. \text{Stream} \bullet sp_1.\text{tr} \Rightarrow sp_2.\text{tr} \\
(p_1 \Rightarrow p_2) & \equiv \forall \Delta. \text{Interval} \bullet p_1.\Delta \Rightarrow p_2.\Delta
\end{align*}
\]

'\( \Leftrightarrow \)' and '\( \equiv \)' are similarly defined with '\( \Rightarrow \)' replaced by '\( \Leftarrow \)' and '\( = \)', respectively.

We let \( \lim_{x} f(x) \) and \( \lim_{x} f(x) \) 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 \( s \in \text{Stream}_V \) [9]. For an expression \( e \in \Sigma \rightarrow \text{Val} \), interval \( \Delta \in \text{Interval} \) and stream \( s \in \text{Stream} \), we define:

\[
\begin{align*}
\forall_e.\Delta.s & \equiv \lim_{t \rightarrow \text{lub} \Delta^{-}} e.s_t \\
\exists_e.\Delta.s & \equiv \lim_{t \rightarrow \text{glb} \Delta^{+}} e.s_t \\
(\downarrow e).\Delta & \equiv \exists \Delta' : \text{Interval} \bullet (\Delta' \propto \Delta) \land \forall_e.\Delta' \\
(\uparrow e).\Delta & \equiv \exists \Delta' : \text{Interval} \bullet (\Delta \propto \Delta') \land \forall_e.\Delta'
\end{align*}
\]

Thus, \( \forall_e \) and \( \exists_e \) 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 \( e \), the everywhere and sometime operators are defined as follows:

\[
\begin{align*}
(\forall_e c).\Delta.s & \equiv \forall \text{tr}. \Delta \bullet c.s_t \\
(\exists_e c).\Delta.s & \equiv \exists \text{tr}. \Delta \bullet c.s_t
\end{align*}
\]

Thus, \( \forall c \) and \( \exists c \) 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 \( p, p_1, p_2 \in \text{IntvPred} \) and
interval $\Delta \in \text{Interval}$ we define:

\[
(p_1; p_2).\Delta \equiv \exists \Delta_1, \Delta_2: \text{Interval} \bullet (\Delta_1 \propto \Delta_2) \land (\Delta = \Delta_1 \cup \Delta_2) \land p_1.\Delta_1 \land p_2.\Delta_2
\]

\[
\neg \neg p.\Delta \equiv \forall \Delta': \text{Interval} \bullet \Delta' \subseteq \Delta \Rightarrow p.\Delta'
\]

\[
\uparrow p.\Delta \equiv \exists \Delta': \text{Interval} \bullet (\Delta \propto \Delta') \land p.\Delta'
\]

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

\[
\begin{align*}
p_1 : p_2 & \equiv p_1 \lor (p_1 ; p_2) \\
\Diamond p & \equiv \neg \Box \neg p \\
\nabla p & \equiv \Diamond p \lor \Box p \\
p_1 \mathsf{un} p_2 & \equiv p_2 \lor (\Box p_1 ; p_2) \lor (\Box p_1 \land \Box (p_1 \lor p_2)) \\
p_1 \mathsf{wu} p_2 & \equiv p_1 \Rightarrow (p_1 \mathsf{un} p_2)
\end{align*}
\]

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

\[
(\Pi p).\Delta.s \equiv p.\text{Time}.s
\]

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

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 $\Delta$ iff $(\mathsf{inv}.c).\Delta$ holds, where

\[
\mathsf{inv}.c \equiv \downarrow c \Rightarrow \Box c
\]

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

\[
\begin{align*}
st.v & \equiv \exists k \bullet \mathsf{inv}.(v = k) \\
st.V & \equiv \forall v \bullet V \bullet st.v
\end{align*}
\]

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 \textit{partition} of an interval. We use $\mathsf{seq}.X$ to denote a possibly infinite sequence with elements of type $X$. A sequence can be explicitly defined using angle brackets, $\langle \cdot \rangle$ and $\langle \langle \cdot \rangle \rangle$, and $\langle \langle \cdot \rangle \rangle$ is the sequence concatenation operator. For a sequence of sets $\sigma$, we define we define $\bigcup \sigma \equiv \bigcup_{i \in \mathsf{dom}.\sigma} \sigma_i$.

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

\[
\begin{align*}
\mathsf{part.}\Delta & \equiv \{ z: \mathsf{seq}.\text{Interval} \mid (\Delta = \bigcup z) \land (\forall i: \mathsf{dom}.z \rightarrow \{0\} \bullet z_{i-1} \propto z_i) \}
\end{align*}
\]

A non-Zeno partition of an $\Delta$ is given by

\[
\begin{align*}
\mathsf{NZpart.}\Delta & \equiv \{ z: \mathsf{part.}\Delta \mid (\mathsf{dom}.z = \mathbb{N}) \Rightarrow (\ell, \Delta = \infty) \}
\end{align*}
\]
Definition 3 (Alternates) For a state predicate c, interval $\Delta \in \text{Interval}$ and a partition $\delta \in \text{part.}\Delta$, we define
\[
\text{alt}.c.\delta \equiv \forall i: \text{dom.}\delta \bullet ((\square c).\delta_i \land (i + 1 \in \text{dom.}\delta)) \land \\
((\square \neg c).\delta_i \land (i + 1 \in \text{dom.}\delta)) \Rightarrow ((\square c).\delta_{i+1})
\]

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

Definition 5 Suppose p is an interval predicate. We say
1. $p$ joins in $\Delta$ iff ($\forall \delta: \text{NZpart.}\Delta \bullet \forall i: \text{dom.}\delta \bullet p.\delta_i$) $\Rightarrow p.\Delta$.
2. $p$ splits in $\Delta$ iff $p.\Delta \Rightarrow \forall \delta: \text{NZpart.}\Delta \bullet (\forall i: \text{dom.}\delta \bullet p.\delta_i)$.

We say $p$ joins and $p$ splits iff $p$ joins in $\Delta$ and $p$ splits in $\Delta$, respectively for any arbitrary interval $\Delta$.

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

Lemma 1 For any state predicate c, interval predicate $\text{inv.c}$ both joins and splits.

The next lemma allows us to perform case analysis to prove formulae of the form $p_1 \Rightarrow p_2$, 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 $p_1 \Rightarrow p_2$ holds provided there exists a non-Zeno state predicate $c$ and both of the following hold:
\[
p_1 \land \square c \Rightarrow p_2 \tag{8}
\]
\[
p_1 \land \square \neg c \Rightarrow p_2 \tag{9}
\]

Proof 1 For an arbitrary interval $\Delta \in \text{Interval}$,
\[
p_1.\Delta
\Rightarrow c \text{ is non-Zeno}
\Rightarrow p_1.\Delta \land \exists \delta: \text{NZpart.}\Delta \bullet \text{alt}.c.\delta
\Rightarrow \text{Definition 5, } p_1 \text{ splits}
\Rightarrow \exists \delta: \text{NZpart.}\Delta \bullet \text{alt}.c.\delta \land \forall i: \text{dom.}\delta \bullet p_1.\delta_i
\Rightarrow (8) \text{ and } (9)
\Rightarrow \exists \delta: \text{NZpart.}\Delta \bullet \forall i: \text{dom.}\delta \bullet p_2.\delta_i
\Rightarrow \text{Definition 5, } p_2 \text{ joins}
\Rightarrow p_2.\Delta
\]

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 $0 < \epsilon_1, \epsilon_2 \in \text{Time}$. Then
\[
p_1 \land \square \epsilon \land (\ell \geq \epsilon_1 + \epsilon_2) \Rightarrow \nabla p_2
\]
holds provided that for some state predicate $c'$, both of the following hold:
\[
p_1 \land \square \epsilon \land (\ell \geq \epsilon_1) \Rightarrow \nabla \square \epsilon \tag{10}
\]
\[
p_1 \land \square \epsilon \land (\ell \geq \epsilon_2) \Rightarrow \nabla p_2 \tag{11}
\]
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.
\[
GP ::= c \rightarrow P \\
P ::= O: \llbracket r, g \rrbracket | \text{seq}\ GP | P \parallel P
\]

An action $O: \llbracket r, g \rrbracket$ 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 $c \rightarrow M$ 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 $M_1 \parallel M_2$ 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.
\[
\begin{align*}
\text{out.}(O: \llbracket r, g \rrbracket) & \triangleq O \\
\text{out.}\langle \rangle & \triangleq \{\} \\
\text{out.}\langle c \rightarrow M \rangle \backsim S & \triangleq \text{out.}M \cup \text{out.}S \\
\text{out.}(M_1 \parallel M_2) & \triangleq \text{out.}M_1 \cup \text{out.}M_2 
\end{align*}
\]

To ensure that the programs we specify are implementable, we define a number of healthiness constraints on the program. The behaviour of any action $O: \llbracket r, g \rrbracket$ may not assume properties of the outputs. Hence we require:
\[
r \in \text{IntvPred}_V \text{ for some } V \subseteq \text{Var}\backslash O \quad \text{for any action } O: \llbracket r, g \rrbracket \tag{12}
\]

For a guarded sequence of programs, we disallow Zeno-like behaviour of the guards. Hence we require:
\[
c \text{ is a non-Zeno state predicate for any program } \langle c \rightarrow M \rangle \backsim S \tag{13}
\]

Finally, two programs executing in parallel may not modify the same outputs. Hence, we require:
\[
\text{out.}M_1 \cap \text{out.}M_2 = \{\} \quad \text{for any program } M_1 \parallel M_2 \tag{14}
\]

3.2 Semantics

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

Figure 1: Guarded sequence and parallel composition
Definition 7 If \( M \) is a teleo-reactive program and \( F \subseteq \text{Var} \) is a set of variables, then:

\[
\text{beh}_F.(O; [r, g]) \triangleq r \Rightarrow g \land \text{st.}(F \setminus O) \quad (15)
\]

\[
\text{beh}_F.() \triangleq \text{true} \quad (16)
\]

\[
\text{beh}_F.T \triangleq ((\llbracket r \cap \text{beh}_F.M \rrbracket : (\llbracket \neg c \land \text{beh}_F.T \rrbracket)) \lor
\quad (\llbracket \neg r \land \text{beh}_F.S \rrbracket : (\llbracket \forall \land \text{beh}_F.T \rrbracket)) \quad (17)
\]

\[
\text{beh}_F.(M_1 \parallel M_2) \triangleq \text{beh}_{F \setminus \text{out} M_2} M_1 \land \text{beh}_{F \setminus \text{out} M_1} M_2 \quad (18)
\]

By (15), the behaviour of an action \( a \), i.e., \( \text{beh}_F.a \) 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 \( \llbracket c \land \text{beh}_F.M \rrbracket \) or \( \llbracket \neg c \land \text{beh}_F.T \rrbracket \) holding initially on the interval. If \( c \) holds initially, either \( c \land \text{beh}_F.M \) holds for the whole interval or the interval may be split into an initial interval in which \( c \land \text{beh}_F.M \) holds, followed by an interval in which \( \neg c \) holds initially and \( \text{beh}_F.T \) holds (recursively) for the second interval. Note that each chopped interval must be a maximal interval over which either \( c \) or \( \neg c \) holds. Note that by (13), \( \text{beh}_F.T \) does not display kinetically 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 \( (c_1 \rightarrow M_1, c_2 \rightarrow M_2) \), 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 \( \neg c_1 \land c_2 \). 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 \( \text{out} M_1 \setminus \text{out} M_2 \) are guaranteed to be stable during execution of \( M_1 \) and similarly, variables \( \text{out} M_2 \setminus \text{out} M_1 \) are guaranteed to be stable during execution of \( M_1 \).

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

Lemma 4 Suppose \( S_1, S_2 \) and \( T \equiv S_1 \land (c \rightarrow M_1) \land S_2 \) are sequences of guarded programs; \( F \subseteq \text{Var} \) is a set of variables; and \( r \) and \( g \) are interval predicates. Then:

\[
\llbracket c \rrbracket \Rightarrow (\text{beh}_F.T = \text{beh}_F.M) \quad (19)
\]

\[
\llbracket \neg c \rrbracket \Rightarrow (\text{beh}_F.T = \text{beh}_F.(S_1 \land S_2)) \quad (20)
\]

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 \( \text{vars}.r \land \text{out}.M = \{\} \); and \( F \supseteq \text{out}.M \) is a set of variables. We define:

\[
F : \{ r \} \ M \{ g \} \triangleq r \land \text{beh}_F.M \Rightarrow g
\]

Theorem 5 \( F : \{ r \} \ O ; [rr, gg] \{ g \} \) holds if \( r \Rightarrow rr \) and \( gg \Rightarrow g \) hold, \( F \supseteq O \) and \( \text{vars}.r \land O = \{\} \).
The next lemma allows us to prove a property of a sequence of guarded programs.

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

\[
\begin{align*}
F: \{r\} & \quad M \quad \{\Rightarrow c \Rightarrow g\} \\
F: \{r\} & \quad S \quad \{\Leftrightarrow c \Rightarrow g\}
\end{align*}
\]

**Lemma 7** Given that \( S_1 \) and \( S_2 \) are sequences of guarded programs, then \( F: \{r\} S_1 \cap \langle c \to M \rangle \cap S_2 \{\Leftrightarrow c \Rightarrow g\} \) holds iff \( F: \{r\} S_1 \cap S_2 \{\Rightarrow c \Rightarrow g\} \) holds.

In program \( M_1 \parallel M_2 \), 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 \( M_1 \parallel M_2 \) (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 \( \text{water}_\text{vol}_\text{rate} \) returns the rate of change of the water level in the tank. Clearly, the pump and hose cannot modify \( \text{water}_\text{vol}_\text{rate} \) simultaneously because the pump makes \( \text{water}_\text{vol}_\text{rate} \) negative while the hose makes \( \text{water}_\text{vol}_\text{rate} \) positive. To resolve this, we may define \( \text{water}_\text{in}_\text{rate} \) (only modified by the hose) and \( \text{water}_\text{out}_\text{rate} \) (only modified by the pump) be the rates at which water is added and removed from the tank, respectively. We may then define \( \text{water}_\text{vol}_\text{rate} = \text{water}_\text{in}_\text{rate} - \text{water}_\text{out}_\text{rate} \).

**Theorem 8** If \( M_1 \parallel M_2 \) is a teleo-reactive program, \( F \supseteq \text{out}.(M_1 \parallel M_2) \) and \( \text{vars}.r_1 \cap \text{out}.M_1 = \text{vars}.(r_2 \wedge g_1) \cap \text{out}.M_2 = \{\} \) then \( F: \{r_1 \wedge r_2\} M_1 \parallel M_2 \{g_1 \wedge g_2\} \) holds provided both of the following hold:

\[
\begin{align*}
F_{\text{out}.M_2}: \{r_1\} & \quad M_1 \quad \{g_1\} \\
F_{\text{out}.M_1}: \{r_2 \wedge g_1\} & \quad M_2 \quad \{g_2\}
\end{align*}
\]

**Proof 2** Because \( M_1 \parallel M_2 \) is a teleo-reactive program, \( \langle \text{in}.M_1 \cup \text{out}.M_1 \rangle \cap \text{out}.M_2 = \{\} \) holds and we have the following calculation:

\[
\begin{align*}
(r_1 \wedge \text{beh}_{F_{\text{out}.M_2}}.M_1) & \Rightarrow g_1 \wedge (r_2 \wedge \text{beh}_{F_{\text{out}.M_1}}.M_2) \Rightarrow (g_1 \Rightarrow g_2) \\
& \Rightarrow \text{logic, weaken antecedents} \\
r_1 \wedge r_2 \wedge \text{beh}_{F_{\text{out}.M_2}}.M_1 \wedge \text{beh}_{F_{\text{out}.M_1}}.M_2 \Rightarrow g_1 \wedge (g_1 \Rightarrow g_2) \\
& \Rightarrow (18), \text{definitions and logic} \\
F: \{r_1 \wedge r_2\} M_1 \parallel M_2 \{g_1 \wedge g_2\}
\end{align*}
\]

**Lemma 9** \( F: \{r_1 \wedge r_2\} M_1 \parallel M_2 \{g_1 \wedge g_2\} \) holds provided both of the following hold:

\[
\begin{align*}
F_{\text{out}.M_2}: \{r_1\} & \quad M_1 \quad \{g_1\} \\
F_{\text{out}.M_1}: \{r_2\} & \quad M_2 \quad \{g_2\}
\end{align*}
\]

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 \( M_1 \parallel M_2 \) denote the simple parallel composition between \( M_1 \) and \( M_2 \). Unlike \( \parallel \), programs under simple parallelism are commutative, i.e., \( \text{beh}_F(M_1 \parallel M_2) = \text{beh}_F(M_2 \parallel M_1) \).

**Lemma 10 (Simple Parallelism)** If \( \text{vars}.r_1 \cap \text{out}.M_2 = \text{vars}.r_2 \cap \text{out}.M_1 = \{\} \) and \( F \supseteq \text{out}.M_1 \cup \text{out}.M_2 \), then

\[
F: \{r_1 \wedge r_2\} M_1 \parallel M_2 \{g_1 \wedge g_2\}
\]
holds provided that both of the following hold:

\[ F_{\text{out}}.M_2: \{r_1\} \rightarrow M_1 \{g_1\} \quad (27) \]
\[ F_{\text{out}}.M_1: \{r_2\} \rightarrow M_2 \{g_2\} \quad (28) \]

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 depot).
move while the robot arm is in the way. The program uses constants

\[
\text{tov}
\]

The following predicates are used to determine specific positions of the table, 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:

\[
G.pos \equiv R_{pos} + (R_{arm,len} + dw \cdot R_{rot})
\]

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.

\[
\begin{align*}
onT.disk & \equiv (disk.pos = T_{pos}) \land (disk.lvl = T.lvl) \\
arrG.disk & \equiv (disk.pos = G.pos) \land (disk.lvl = R_{lvl}) \\
ind.disk & \equiv (disk.pos = D_{pos}) \land (disk.lvl = 0) \\
hbR.disk & \equiv atG.disk \land (G.dist = dw)
\end{align*}
\]

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.

\[
\begin{align*}
T_{at\_FB} & \equiv T.lvl = FB_{lvl} \\
T_{at\_R} & \equiv T.lvl = R_{lvl} \\
full & \equiv \exists disk: Disk \bullet onT.disk \\
holding & \equiv \exists disk: Disk \bullet hbR.disk \\
G_{at\_T} & \equiv G.pos = T_{pos} \\
G_{at\_D} & \equiv G.pos = D_{pos} \\
G_{open} & \equiv G.dist = max_G \\
G_{near\_T} & \equiv G.pos \in T_{area}
\end{align*}
\]

Thus, \( T_{at\_FB} \) holds if 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 \( G_{near\_T} \) 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
it is full and not yet at the robot level. Execution of runT raises the table by executing Raise while \( \neg (\text{full} \land \neg \text{T at } R) \land (\neg \text{full} \land \neg \text{T at } FB) \) holds, which simplifies to \( \neg \text{full} \land \neg \text{T at } FB \). 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 \( \neg \text{GnearT} \) 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:

\[
TR \triangleq \text{Table} \parallel \text{Robot}
\]

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:

\[
TR: \{\text{true}\} \quad TR: \{\text{inv}.(\text{GnearT} \Rightarrow T_{\mu}R)\}
\]  

(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.\( (\text{GnearT} \Rightarrow T_{\mu}R) \) is modified by both Table and Robot. Instead, we obtain the following calculation:

\[
\begin{align*}
(35) & \Leftarrow \text{logic} \quad TR: \{\text{true}\} \quad TR: \{\text{inv}.(\text{GnearT} \Rightarrow T_{\mu}R)\} \\
& \Leftarrow \text{Lemma 7} \quad TR: \{\text{true}\} \quad \langle \text{holding} \Rightarrow \text{drop\_at\_depot}, \text{true} \Rightarrow \text{Rot}\_mid \rangle \\
& \quad \{\text{inv}.(\text{GnearT} \Rightarrow T_{\mu}R) \Rightarrow \downarrow(\text{GnearT} \land \neg T_{\mu}R)\}
\end{align*}
\]
= logic

\[ TR: \{true\} Nil \models \langle \text{holding} \rightarrow \text{drop\_at\_depot}, \text{true} \rightarrow \text{Rot\_mid} \rangle \{ \text{inv} (\text{GnearT} \Rightarrow T\_at\_R) \} \]

\[ \Leftarrow \text{Lemma 9} \]

\[ T: [true] Nil \{ st(T.lvl) \} \land R: [true] \langle \text{holding} \rightarrow \text{drop\_at\_depot}, \text{true} \rightarrow \text{Rot\_mid} \rangle \{ \text{st}(T.lvl) \Rightarrow \text{inv} (\text{GnearT} \Rightarrow T\_at\_R) \} \]

\[ \Leftarrow \text{first triple: Theorem 5} \]

\[ \text{second triple: logic, use st}(T.lvl) \]

\[ R: [true] \langle \text{holding} \rightarrow \text{drop\_at\_depot}, \text{true} \rightarrow \text{Rot\_mid} \rangle \{ \text{inv} (\lnot \text{GnearT}) \} \]

\[ \Leftarrow \text{Theorem 6 twice} \]

\[ R: [true] \text{Ungrip} \{ \text{holding} \land \Box G\_at\_D \Rightarrow \text{inv} (\lnot \text{GnearT}) \} \]

\[ R: [true] \text{Rot\_dep} \{ \text{holding} \land \Box (\lnot G\_at\_D) \Rightarrow \text{inv} (\lnot \text{GnearT}) \} \]

\[ R: [true] \text{Rot\_mid} \{ \Box \lnot \text{holding} \Rightarrow \text{inv} (\lnot \text{GnearT}) \} \]

\[ \Leftarrow \Box G\_at\_D \Rightarrow \Box \lnot \text{GnearT} , \text{beh}_R.R\_dep \lor \text{beh}_R.R\_mid \Rightarrow \text{inv} (\lnot \text{GnearT}) \]

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.,

\[ \{ r_1 \land (\ell \geq \epsilon) \} \quad TR \quad \lnot \text{onT}\_\text{disk} \Rightarrow \lnot \text{hbR}\_\text{disk} \quad (36) \]

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

\[ \{ r_2 \land (\ell \geq \kappa) \} \quad TR \quad \text{hbR}\_\text{disk} \Rightarrow \lnot \text{inD}\_\text{disk} \quad (37) \]

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).

\( (36) \)

\[ \Leftarrow \text{Definition 8 and logic} \]

\[ \{ r_1 \land (\ell \geq \epsilon) \} \quad TR \quad \lnot \text{onT}\_\text{disk} \land \Box \lnot \text{hbR}\_\text{disk} \Rightarrow \lnot \text{hbR}\_\text{disk} \]  

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

\[ r_1 \Rightarrow \forall T.lvl, R.rot, G\_dist \bullet \lnot \text{onT}\_\text{disk} \land \Box \lnot \text{hbR}\_\text{disk} \Rightarrow \Box \text{onT}\_\text{disk} \]

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:

\[ \{ r_1 \land (\ell \geq \epsilon) \} \quad TR \quad \Box \text{onT}\_\text{disk} \Rightarrow \lnot \text{hbR}\_\text{disk} \]

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:

\[ \{ r_1 \land (\ell \geq \epsilon_1) \} \quad TR \quad \Box \text{onT}\_\text{disk} \Rightarrow \lnot \text{R\_at\_R} \quad (38) \]

\[ \{ r_1 \land (\ell \geq \epsilon_2) \} \quad TR \quad \Box \text{onT}\_\text{disk} \land \lnot \text{R\_at\_R} \Rightarrow \lnot \text{hbR}\_\text{disk} \quad (39) \]

12
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:

\[
\{ \text{true} \} \quad \text{TR} \quad \{ \text{inv} : (R_{\text{lvl}} \leq T_{\text{lvl}} \leq FB_{\text{lvl}}) \}
\]

which is an easily provable safety condition.

**Proof of (38).**

\[
\begin{align*}
\{ r_1 \land (\ell \geq \epsilon_1) \} & \quad \text{TR} \quad \{ \square \text{onT.disk} \Rightarrow \nabla \nabla \text{at}_R \} \\
\leq & \quad \text{logic, } \square (\text{onT.disk} \Rightarrow \text{full}) \\
\{ r_1 \land (\ell \geq \epsilon_1) \} & \quad \text{TR} \quad \{ \square (\text{full} \land \nabla \text{at}_R) \Rightarrow \nabla \text{at}_R \} \\
\leq & \quad (35), \text{parallel composition (18)} \\
\{ r_1 \land (\ell \geq \epsilon_1) \} & \quad \text{Table} \quad \{ \square (\text{full} \land \nabla \text{at}_R \land \neg \text{GnearT}) \Rightarrow \nabla \text{at}_R \} \\
\leq & \quad (19) \text{ and (20)} \\
\{ r_1 \land (\ell \geq \epsilon_1) \} & \quad \text{Lower} \quad \{ \square (\text{full} \land \nabla \text{at}_R \land \neg \text{GnearT}) \Rightarrow \nabla \text{at}_R \} \\
\leq & \quad (31) \text{ (i.e., definition of Lower), (40) and assumption } r_1 \text{ true}
\end{align*}
\]

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

\[
\{ \text{true} \} \quad \text{Table} \quad \{ \nabla \text{at}_R \ \text{wu} \ \neg \text{full} \}
\]

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:

\[
\begin{align*}
(39) & \quad \text{true} \quad \text{using (41)} \\
\{ r_1 \land (\ell \geq \epsilon_2) \} & \quad \text{TR} \quad \{ \square (\text{onT.disk} \land \text{at}_R) \Rightarrow \nabla \text{holding} \}
\end{align*}
\]

As before, we can now assume the table remains at the robot level throughout the interval as opposed to only at the start. Assuming \( \epsilon_2 = \epsilon_2_1 + \epsilon_2_2 \), we apply Lemma 3 (transitivity) to obtain the following cases:

\[
\begin{align*}
\{ r_1 \land (\ell \geq \epsilon_2_1) \} & \quad \text{TR} \quad \{ \square (\text{onT.disk} \land \text{at}_R) \Rightarrow \nabla \text{holding} \} \\
\{ r_1 \land (\ell \geq \epsilon_2_2) \} & \quad \text{TR} \quad \{ \square (\text{onT.disk} \land \text{at}_R \land \neg \text{holding}) \Rightarrow \nabla \text{holding} \}
\end{align*}
\]

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:

\[
\{ \text{true} \} \quad \text{Robot} \quad \{ \square \text{full} \land \nabla \neg \text{holding} \Rightarrow \square \neg \text{holding} \}
\]

then obtain the following calculation:

\[
\begin{align*}
(43) & \quad (44) \text{ because onT disk} \Rightarrow \text{full} \\
\{ r_1 \land (\ell \geq \epsilon_2_2) \} & \quad \text{TR} \quad \{ \square (\text{onT.disk} \land \text{at}_R \land \neg \text{holding}) \Rightarrow \nabla \text{holding} \}
\end{align*}
\]

The rely condition above states that the interval is of length \( \epsilon_2_2 \) 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 $m \subseteq \text{out}.M$, we use $M \setminus m$ to denote a program in which $m$ is hidden from the environment. The outputs of program $\text{out}.(M \setminus m)$ is defined as:

$$\text{out}.(M \setminus m) \equiv \text{out}.M \setminus m$$

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

$$\text{beh}_{F\setminus m}(M \setminus m) \equiv \exists m \ast \text{beh}_{F}.M$$ (45)

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

Theorem 11 (Hiding) If $m \subseteq \text{out}.M$, $F \supseteq \text{out}.M$ and $F: \{r\} \text{M}\{g\}$, then $F \setminus m: \{r\} M \setminus m \{\exists m \ast g\}$.

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

$$F \setminus m: \{r\} M \setminus m \{\exists m \ast g\}$$
$$= \text{expand triple, (45)}$$
$$r \land (\exists m \ast \text{beh}_{F}.M) \Rightarrow \exists m \ast g$$
$$\Leftarrow m \not\in r$$
$$(\exists m \ast r \land \text{beh}_{F}.M) \Rightarrow \exists m \ast g$$
$$\Leftarrow \text{logic}$$
$$F: \{r\} \text{M}\{g\} \square$$

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.,

$$\text{out}.(\mu e^h \bullet M) \equiv \text{out}.M \cup e$$

This means that the rely condition of $\mu e^h \bullet M$ 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:

$$\text{beh}_{F}.(\mu e^h \bullet M) \equiv (\text{beh}_{F}.M)[e^h]$$ (46)

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

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

14
We provide a concrete example by considering an oscillator that is constructed using an inverter.

### Lemma 13

**Proof 5**

\[ F; \{ r \} \mu e ; h \cdot M \{ g[e \setminus h]\} \]

\[ = \]

**definitions**

\[ r_1 \land \text{beh}_F.M[e \setminus h] \models g[e \setminus h] \]

\[ \Leftarrow \]

**assumption:** \( r_1 \land \text{beh}_F.M[e \setminus h] \models r[e \setminus h] \]

\[ r[e \setminus h] \land \text{beh}_F.M[e \setminus h] \models g[e \setminus h] \]

\[ = \]

**logic**

\[ (r \land \text{beh}_F.M \models g)|e\setminus h] \]

\[ \Leftarrow \]

**assumption:** \( F; \{ r \} M \{ g \} \)

true

\[ \square \]

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 feedback inputs \( e \) replaced by outputs \( h \).

The lemma below states that replacing a component \( M \) by a component \( M' \equiv \mu e ; h \cdot M \) within a guarded program \( T \equiv \langle c \rightarrow M \rangle \land S \), then the behaviour of \( \mu e ; h \cdot T \) is equivalent to the program \( \mu e ; h \cdot \langle c \rightarrow M' \rangle \land S \).

**Lemma 13** If \( T \equiv \langle c \rightarrow M \rangle \land S \), \( T' \equiv \langle c \rightarrow \mu e \cdot h \cdot M \rangle \land S \) and \( F \supseteq \text{out}.T \) then

\[ \text{beh}_F.(\mu e ; h \cdot T') \equiv \text{beh}_F.(\mu e ; h \cdot T) \]

### Proof 5

\[ \text{beh}_F.(\mu e ; h \cdot T'), \Delta \]

\[ = \]

**definition of feedback**

\[ \text{beh}_F.(T'[e \setminus h]), \Delta \]

\[ = \]

**logic**

\[ \exists \delta : \text{NZpart}. \Delta \bullet \forall i : \text{dom} \cdot \delta \bullet ((\Box c \land \text{beh}_F.(\mu e ; h \cdot M))[e \setminus h]) \cdot \delta_i \lor (\Box \neg c \land \text{beh}_F.S)[e \setminus h]) \cdot \delta_i \]

\[ \exists \delta : \text{NZpart}. \Delta \bullet \forall i : \text{dom} \cdot \delta \bullet ((\Box c \land \text{beh}_F.M)[e \setminus h]) \cdot \delta_i \lor (\Box \neg c \land \text{beh}_F.S)[e \setminus h]) \cdot \delta_i \]

\[ = \]

**beh definition**

\[ (\text{beh}_F.T)[e \setminus h]), \Delta \]

\[ = \]

**beh definition**

\[ \text{beh}_F.(\mu e ; h \cdot T), \Delta \]

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

\[ \text{beh}_F.\text{inv} \equiv \forall t : \text{Time} \cdot (t < c \Rightarrow \neg \text{on} @ t) \land (\text{on} @ (t + \epsilon) = \neg \text{on}_c @ t) \]
Now, given the following rely condition:

\[
\text{rely, } \Delta \triangleq \exists \delta : \text{NZpart, } \Delta \bullet (\forall \ell : \text{dom } \delta \bullet \ell. \delta_i = \epsilon) \land \text{alt on, } \delta \land \neg \text{on, } \delta_0
\]

which states that the value of on flips after every \(\epsilon\) time units, we have

\[
F : \{\text{rely}\} \text{ inv } \{\text{rely[on, on]}\}
\]

That is, given that the value of input on oscillates every \(\epsilon\) 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. That is, we define

\[
\text{osc } \triangleq \mu \text{ on } \bullet \text{ inv}
\]

We prove our desired property of the oscillator:

\[
F : \{\text{true}\} \text{ osc } \{\text{rely[on, on]}\}
\]

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

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 feedback 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 \(M_1 \gg M_2\) to denote the pipeline from \(M_1\) to \(M_2\) (see Fig. 6), which is defined as follows:

\[
M_1 \gg M_2 \triangleq (M_1 \parallel M_2) \setminus \text{out} \cdot M_1
\]

hence, we have

\[
\text{out}(M_1 \gg M_2) = \text{out} \cdot M_2
\]

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 (Pipepline)** If \(\text{out} \cdot M_1 \cap \text{vars}(r_1 \land r_2) = \text{out} \cdot M_1 \cap g = \{\}\), then

\[
F \setminus \text{out} \cdot M_1 : \{r_1 \land r_2\} M_1 \gg M_2 \{g\}
\]

holds provided that both of the following hold:

\[
F \setminus \text{out} \cdot M_2 : \{r_1\} M_1 \{g_1\}
\]

\[
F \setminus \text{out} \cdot M_1 : \{r_2 \land g_1\} M_2 \{g\}
\]

**Proof 6**

\[
F \setminus \text{out} \cdot M_1 : \{r_1 \land r_2\} M_1 \gg M_2 \{g\}
\]

(48) and definitions

\[
F \setminus \text{out} \cdot M_1 : \{r_1 \land r_2\} (M_1 \parallel M_2) \setminus \text{out} \cdot M_1 \{g\}
\]

Theorem 11, out \cdot M_1 does not occur free in \(r_1 \land r_2\) and \(g\)

\[
F : \{r_1 \land r_2\} M_1 \parallel M_2 \{g\}
\]

\[
\Leftrightarrow \quad \text{Theorem 8 with } g_2 \text{ replaced by } g
\]

(49) \land (50) \hspace{1cm} \Box
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


