Fractional Permissions and Non-Deterministic Evaluators in Interval Temporal Logic

Brijesh Dongol, John Derrick, Ian J. Hayes

18 pages
Fractional Permissions and Non-Deterministic Evaluators in Interval Temporal Logic

Brijesh Dongol\(^1\), John Derrick\(^2\)*, Ian J. Hayes \(^3\)†

\(^1\) B.Dongol@sheffield.ac.uk, \(^2\) J.Derrick@dcs.shef.ac.uk
Department of Computer Science,
The University of Sheffield
\(^3\) Ian.Hayes@itee.uq.edu.au
School of Information Technology and Electrical Engineering
The University of Queensland
2012-06-06 16:09

Abstract: We propose Interval Temporal Logic as a basis for reasoning about concurrent programs with fine-grained atomicity due to the generality it provides over reasoning with standard pre/post-state relations. To simplify the semantics of parallel composition over intervals, we use fractional permissions, which allows one to ensure that conflicting reads and writes to a variable do not occur simultaneously. Using non-deterministic evaluators over intervals, we enable reasoning about the apparent states over an interval, which may differ from the actual states in the interval. The combination of Interval Temporal Logic, non-deterministic evaluators and fractional permissions results in a generic framework for reasoning about concurrent programs with fine-grained atomicity. We use our logic to develop rely/guarantee-style rules for decomposing a proof of a large system into proofs of its subcomponents, where fractional permissions are used to ensure that the behaviours of a program and its environment do not conflict.

Keywords: Interval Temporal Logic, Fractional Permissions, Non-deterministic Expression Evaluation, Parallel Composition, Compositional Reasoning

1 Introduction

Current frameworks for reasoning about shared-variable concurrency are based on interleaving models (e.g., [OG76, Jon83, STER11]) or are an extension Hoare’s methods for reasoning about sequential programs [Hoa69] (e.g., separation logic [Rey02]). A common aspect of these frameworks is that they consider relations between the pre- and post-states of a program transition. In the context of modern multicore/multiprocessor systems that use so-called relaxed memory models, a relational view is not always adequate because it is possible for concurrent processes to execute in a truly concurrent manner (e.g., by threads that are executed in another processor core). Thus, program verification poses a difficult challenge because one is often forced to consider the internal behaviour of a command. In particular, one cannot rely on stability of the

* Brijesh Dongol and John Derrick are sponsored by EPSRC Grant EP/J003727/1.
† Ian J. Hayes is sponsored by ARC Discovery Grant DP0987452.
shared variables from the pre-state of a transition because variables may be modified while an
expression is being evaluated [HBDJ11]. Our goals for this paper are not to verify the low-
level relaxed memory models [SVN+ 11]. Instead, we aim to develop a framework for verifying
(high-level) programs in which state predicates may not be evaluated atomically.

Example 1 Consider the parallel program (if u < v then S₁ else S₂ fi)(u := 1; v := 1), where
the if and the assignment statements are executed by processes x and y, respectively. Assume that
the initial state of the program satisfies u, v = 0, 0. Using an execution model that only considers
pre/post-state relations, process x will always execute S₂ because the state in which guard u < v
is evaluated satisfies either u, v = 0, 0 (the initial state), u, v = 1, 0 (the state after y executes u := 1)
or u, v = 1, 1 (the state after y executes v := 1). This execution model does not accurately reflect
the real world where it is possible for values of variables to change while an expression is being
evaluated [CJ07, HBDJ11]. In particular, u could be read in the initial state (which satisfies
u, v = 0, 0) and v could be read after both variables have changed (i.e., in a state that satisfies
u, v = 1, 1), and hence there is an apparent state that satisfies u, v = 0, 1, and it is possible for
process x to execute S₁.

In this paper, we use Interval Temporal Logic [Mos00] to reason about a program’s behaviour
within the (discrete) interval of time in which the program executes. By combining interval-
based reasoning with logics for non-deterministic evaluation [HBDJ11], we develop a framework
that allows reasoning about both the actual and apparent states within an observation interval.
Hence, our framework accurately captures the possible fine-grained interleaving between parallel
components. In particular, we are able to identify that, in Example 1 it is possible for process x
to execute S₁ (see Section 4.5.2).

An advantage of using Interval Temporal Logic is that the behaviour of the parallel compo-
sition of concurrent components may be defined to be the conjunction of the behaviour of each
component. Clearly, this causes no real problems if the sets of variables of the parallel compo-
nents are pairwise disjoint. However, if this is not the case, strong synchronisation requirements
are sometimes assumed. For example, Cau and Zedan require that any output of any component
is simultaneously (i.e., in the same state) read by another [CZ97], which is unrealistic. In reality,
a process that is writing to a location should prevent other parallel processes from writing to
and reading from the same location simultaneously [Boy03]. For example, with the program in
Example 1, process x should not read u in the transition in which process y updates the value
of u to 1. To formalise conflicting behaviours, we incorporate fractional permissions [Boy03]
into our framework, which allows read/write access to program variables to be controlled in a
straightforward manner. Fractional permissions have been incorporated into separation logic to
ensure that variables are not simultaneously modified [BCOP05].

To the best of our knowledge, a logic that combines Interval Temporal Logic [Mos00], non-
deterministic expression evaluation [HBDJ11] and fractional permissions [Boy03] as a method
for reasoning about fine-grained atomicity within truly concurrent programs is novel. In addition,
we develop a number of rules that enable decomposition of a proof of a larger component into
proofs of the subcomponents.

This paper is structured as follows. In Section 2 we present the theory of interval predicates
and methods for non-deterministically evaluating state predicates over an interval. We present
our treatment of fractional permissions in manner that allows integration with Interval Temporal Logic in Section 3. In Section 4 we present commands, their formalisation using interval predicates and their refinement. Section 5 presents compositional methods for reasoning about commands.

2 Interval predicates and non-deterministic evaluation

2.1 Interval predicates

An interval is a contiguous set of integers (denoted \(Z\)), and hence the set of all intervals is:

\[
\text{Intv} \equiv \{ \Delta \subseteq Z | \forall t,t': \Delta \cdot \forall u: Z \cdot t \leq u \leq t' \Rightarrow u \in \Delta \}
\]

Using ‘.’ for function application, we let \(\text{lub}\) and \(\text{glb}\) denote the least upper and greatest lower bounds of an interval \(\Delta\), respectively, and we define \(\text{lub}\emptyset \equiv -\infty\) and \(\text{glb}\emptyset \equiv \infty\). If the size of \(\Delta\) is infinite and \(\text{glb}\Delta \in Z\), then \(\text{lub}\Delta = \infty\) (i.e., is not a member of \(Z\)) and if \(\Delta\) is infinite and \(\text{lub}\Delta \in Z\) then \(\text{glb}\Delta = -\infty\).

We must often reason about two adjoining intervals, i.e., intervals that immediately precede or follow a given interval. For \(\Delta, \Delta' \in \text{Intv}\), we define

\[
\Delta \diamond \Delta' \equiv \Delta \neq \emptyset \land \Delta' \neq \emptyset \Rightarrow (\text{lub}\Delta < \text{glb}\Delta') \land (\Delta \cap \Delta' = \emptyset) \land (\Delta \cup \Delta' \in \text{Intv})
\]

Thus, \(\Delta \diamond \Delta'\) holds if and only if \(\Delta'\) immediately follows \(\Delta\). By conjunct \(\Delta \cap \Delta' = \emptyset\), adjoining intervals \(\Delta\) and \(\Delta'\) must be disjoint, and by conjunct \(\Delta \cup \Delta' \in \text{Intv}\), the union of \(\Delta\) and \(\Delta'\) must be contiguous. Note that both \(\Delta \diamond \emptyset\) and \(\emptyset \diamond \Delta\) hold trivially.

We use \(\text{Empty}\Delta\) to denote that the given interval \(\Delta\) is empty.

\[
\text{Empty}\Delta \equiv \Delta = \emptyset
\]

Given that variable names are taken from the set \(\text{Var}\), a state space over a set of variables \(V \subseteq \text{Var}\) is given by \(\text{State}_V \equiv V \rightarrow \text{Val}\) and a state is a member of \(\text{State}_V\), i.e., a state is a total function mapping variables in \(V\) to values in \(\text{Val}\). Note that we do not distinguish between variables and memory heap (e.g., as done in [Boy03, BCOP05]), however, our methods can be extended to explicitly distinguish between the two concepts in a straightforward manner.

A stream of behaviours over \(V\) is given by the total function \(\text{Stream}_V \equiv Z \rightarrow \text{State}_V\), which maps each time in \(Z\) to a state over \(V\). A predicate over type \(T\) is a total function \(\text{P}T \equiv T \rightarrow \mathbb{B}\) mapping each member of \(T\) to a Boolean. For example \(\text{PState}_V\) and \(\text{PStream}_V\) denote state and stream predicates, respectively. To facilitate reasoning about specific parts of a stream, we use interval predicates, which have type \(\text{IntvPred}_V \equiv \text{Intv} \rightarrow \text{PStream}_V\).

We assume pointwise lifting of operators on stream and interval predicates in the normal manner, e.g., if \(p_1\) and \(p_2\) are interval predicates, \(\Delta\) is an interval and \(s\) is a stream, we have \((p_1 \land p_2)\cdot \Delta.s = (p_1\cdot \Delta.s \land p_2\cdot \Delta.s)\). When reasoning about properties of programs, we would like to state that whenever a property \(p_1\) holds over any interval \(\Delta\) and stream \(s\), a property \(p_2\) also holds over \(\Delta\) and \(s\). Hence, we define universal implication for \(p_1, p_2 \in \text{IntvPred}_V\) as

\[
p_1 \Rightarrow p_2 \equiv \forall \Delta: \text{Intv}, s: \text{Stream}_V \cdot p_1, \Delta.s \Rightarrow p_2, \Delta.s
\]

We say \(p_1 \equiv p_2\) holds iff both \(p_1 \Rightarrow p_2\) and \(p_2 \Rightarrow p_1\) hold.
2.2 Evaluating state predicates over intervals

Most implementations only guarantee that at most one global variable can be read in a single atomic step. Thus, in the presence of possibly interfering processes and fine-grained atomicity, an evaluation model that assumes state predicates can be evaluated in a single state may not be implementable [CJ07, JP11]. That is, although an implementation evaluates a state predicate using a number of fine-grained atomic steps, this reality is not reflected in a model that assumes coarse-grained atomicity. Hence, we consider methods for non-deterministically evaluating state predicates over an observation interval [HBDJ11]. To this end, we consider the sets of states and sets of apparent states evaluators, which we introduce using Examples 2 and 3, respectively. The evaluators are formalised using functions states and apparent, which for an interval \( \Delta \) and stream \( s \) are defined below:

\[
\text{states.}\Delta.s \triangleq \{s' : \text{State}_V \mid \exists r : \Delta \cdot r = s.t\}
\]

\[
\text{apparent.}\Delta.s \triangleq \{s' : \text{State}_V \mid \forall v : \exists r : \Delta \cdot r = s.t \cdot \sigma \}
\]

**Example 2 (Sets of states)** Consider the statements \( u := 1 \); \( v := 1 \) from Example 1 which we assume are executed over an interval \( \Delta \) from an initial state that satisfies \( u, v = 0, 0 \). Assuming no other (parallel) modifications to \( u \) and \( v \), for some stream \( s \) over \( \{u, v\} \), the set of actual states that the assignments generate are:

\[
\text{states.}\Delta.s = \left\{ \{u \mapsto 0, v \mapsto 0\}, \{u \mapsto 1, v \mapsto 0\}, \{u \mapsto 1, v \mapsto 1\} \right\}.
\]

The sets of states approach evaluates the given state predicate in one of the actual states of the observation interval. For example, \( u \leq v \) evaluated in \( s \) within \( \Delta \) above has possible values \{false, true\}. Both \( u < v \) and \( v = v \) only have a single possible value, false and true, respectively.

Although the sets of states approach takes the non-determinism that results from concurrent executions into account, the approach does not yet reflect the fact that most implementations will only be able to read at most one variable atomically. Thus, we consider a second approach in which at most one variable is read in each state of the observation interval. However, each free variable of a state predicate is read at most once, and the same value is used for each occurrence of the variable within the state predicate\(^1\).

**Example 3 (Sets of apparent states.)** The set of apparent states corresponding to Example 1 is\(^2\):

\[
\text{apparent.}\Delta.s = \left\{ \{u \mapsto 0, v \mapsto 0\}, \{u \mapsto 1, v \mapsto 1\}, \{u \mapsto 0, v \mapsto 1\}, \{u \mapsto 1, v \mapsto 0\} \right\}.
\]

Note that \( \text{apparent.}\Delta.s = \text{states.}\Delta.s \cup \{\{u \mapsto 0, v \mapsto 1\}\} \). Using a sets of apparent states evaluation, a predicate is evaluated in one of the states of \( \text{apparent.}\Delta.s \). Hence, unlike the sets of states approach, the possible values of both \( u \leq v \) and \( u < v \) are \{false, true\}. However, \( v = v \) still only has one possible value, true, because the same value of \( v \) is used for both occurrences of \( v \).

\(^1\) It is possible to define evaluators that, for example, (re)read a variable for each occurrence of the variable, and hence, potentially returns false for \( v = v \) if the value of \( v \) changes during the observation interval [CJ07, JP11, HBDJ11].

\(^2\) Note that the apparent states within an interval may differ from the Cartesian product between variables in \( V \) and their values in the interval.
Two useful operators for a sets of states evaluation of a state predicate \( c \) are \( 
abla c \) and \( 
abla c \), which ensure that \( c \) holds in all and some actual state of the given stream within the given interval, respectively. Similarly, two useful operators for a sets of apparent states evaluation allow one to formalise that \( c \) definitely holds (denoted \( @c \)) and \( c \) possibly holds (denoted \( \boxdot c \)) in all apparent states of a stream within an interval\(^3\).

\[
\begin{align*}
(\nabla c).\Delta s & \equiv \forall \sigma. \text{states}.\Delta s \cdot c.\sigma \\
(\nabla c).\Delta s & \equiv \exists \sigma. \text{states}.\Delta s \cdot c.\sigma \\
(\boxdot c).\Delta s & \equiv \forall \sigma. \text{apparent}.\Delta s \cdot c.\sigma \\
(\boxdot c).\Delta s & \equiv \exists \sigma. \text{apparent}.\Delta s \cdot c.\sigma
\end{align*}
\]

The following lemma states a relationship between definitely and always properties, as well as between possibly and sometime properties [HBDJ11].

**Lemma 1** Both \( @c \Rightarrow \boxdot c \) and \( \boxdot c \Rightarrow @c \) hold, but the converses of both implications are not necessarily true.

We say a variable \( v \) is stable in an interval \( \Delta \) (denoted \( \text{st}.v.\Delta \)) iff the value of \( v \) does not change from its value at the start of \( \Delta \). A set of variables \( V \) is stable in \( \Delta \) (denoted \( \text{st}.V.\Delta \)) iff each variable in \( V \) is stable in \( \Delta \). For a stream \( s \), we define:

\[
\begin{align*}
(\text{st}.v).\Delta s & \equiv \text{glb}.\Delta \in \mathbb{Z} \land \forall (v = s.(\text{glb}\cdot\Delta - 1).v),\Delta \\
\text{st}.V & \equiv \forall v; V \cdot \text{st}.v
\end{align*}
\]

Note that every variable is stable in an empty interval and the empty set of variables is stable in any interval, i.e., both \( \text{st}.V.\emptyset \) and \( \text{st}.\emptyset.\Delta \) hold trivially.

We let \( \text{vars}.c \) denote the free variables of state predicate \( c \). The following lemma states that if all but one variable of \( c \) is stable over an interval \( \Delta \), then \( c \) definitely holds in \( \Delta \) iff \( c \) always holds in \( \Delta \) and \( c \) possibly holds in \( \Delta \) iff \( c \) holds sometime in \( \Delta \) [HBDJ11].

**Lemma 2** For any state predicate \( c \) and variable \( v \), \( \text{st}.(\text{vars}.c \setminus \{v\}) \Rightarrow (\nabla c = \boxdot c) \land (\boxdot c = \nabla c) \).

If every free variable of a state predicate is stable over the evaluation interval, then the evaluators defined above are equivalent. Lemmas 1 and 2 are used to prove Lemma 3 below, which in turn is used to prove Theorem 1.

**Lemma 3** For any state predicate \( c \), \( \text{st}.(\text{vars}.c) \land \neg \text{Empty} \Rightarrow (\nabla c = \boxdot c = \boxdot c = \nabla c) \).

### 3 Fractional permissions

#### 3.1 Read/write permissions and interference

As we shall see in Section 4.3, the behaviour a process executing a command is formalised by an interval predicate, and the behaviour of a parallel execution over an interval is given by the conjunction of these behaviours over the same interval. Because the state-spaces of the two processes are not disjoint, there is a possibility that a process writing to a variable conflicts with

\(^3\) Our notation follows Burns and Hayes [BH10] and should not be confused with modal operator ‘always’ (\( \boxdot \)) and ‘next’ (\( \nabla \)). Instead, focus on the ‘\( \ast \)’ within \( \nabla \) and \( \nabla \), which represents “for all” and ‘\( \ast \)’ within \( \nabla \) and \( \nabla \), which represent “for some” as used when writing regular expressions.
a read or write to the same variable by another process. To ensure that such conflicts do not take place, we follow Boyland’s idea of mapping variables to a fractional permission [Boy03], which is rational number in the interval $[0,1]$. A process has write-only access to a variable if its permission to access $v$ is 1, has read-only access to $v$ if its permission to access $v$ is above 0 but below 1, and has no access to $v$ if its permission to access $v$ is 0. Note that a process may not have both read and write permission to a variable. Because a permission is a rational number, read access to a variable may be split arbitrarily (including infinitely) among the processes of the system. However, at most one process may have write permission to a variable in any given state. Note that the precise value of the read permission is not important, i.e., there is no notion of priority among processes based on the values of their read permissions.

We let $\text{Proc}$ denote the set of all process identifiers and assume the permission for $x \in \text{Proc}$ to access $v \in V \subseteq \text{Var}$ in $\sigma \in \text{State}_V$ is given by $\Pi_{x,v,\sigma}$. We formalise write, read and no permissions as follows.

**Definition 1** A process $x \in \text{Proc}$ has write-permission to variable $v$ in state $\sigma$ iff $\Pi_{x,v,\sigma} = 1$, has read-permission to $v$ in $\sigma$ iff $0 < \Pi_{x,v,\sigma} < 1$, and has no-permission to access $v$ in $\sigma$ iff $\Pi_{x,v,\sigma} = 0$.

Note that unlike some approaches (e.g., [BCOP05, RG09]), where $x$ has read access to $v$ in state $\sigma$ if $\Pi_{x,v,\sigma} \leq 1$, we explicitly model $\Pi_{x,v,\sigma} = 0$ to mean “no access”. This level of control is necessary because reads and writes to a variable may occur in a truly concurrent manner.

We introduce the following shorthands, which define state predicates for a process $x$ to have read-only and write-only permissions to a variable $v$, and to be denied permission to access $v$.

\[
\begin{align*}
\text{R}_x.v & \equiv 0 < \Pi_{x,v,\sigma} < 1 \\
\text{W}_x.v & \equiv \Pi_{x,v,\sigma} = 1 \\
\text{D}_x.v & \equiv \Pi_{x,v,\sigma} = 0
\end{align*}
\]

In the context of a stream $s$, for any time $t \in \mathbb{Z}$, process $x$ may only write to and read from $v$ in the transition step from $s.(t-1)$ to $s.t$ if $\text{W}_x.v.(s.t)$ and $\text{R}_x.v.(s.t)$, respectively. Thus, $\text{W}_x.v.(s.t)$ does not grant process $x$ permission to write to $v$ in the transition from $s.t$ to $s.(t+1)$ (and similarly $\text{R}_x.v.(s.t)$).

We may use fractional permissions to characterise interference within a process. One must often define a closed system that consists of a number of parallel processes. Hence, for a non-empty set of processes $X$, we define the following, which states that there may be interference on $v$ from a process not in $X$.

\[
I_X.v \equiv \exists y: \text{Proc}\setminus X \cdot \text{W}_y.v
\]

Such notions are particularly useful because we aim to develop rely/guarantee-style reasoning, where we use rely conditions to characterise the behaviour of the environment. We illustrate how permissions are used to formalise the behaviours of a program using the two concrete examples in Section 4.5.

### 3.2 Healthiness conditions

In this section, we introduce healthiness conditions on streams using fractional permissions that formalise our assumptions on the underlying hardware. Below, we assume that the local variables of set of processes $X$ are members of the set $L\text{Var}_X \subseteq \text{Vars}$.
If no process has write access to \( v \) then the value of \( v \) does not change in the transition from \( s.t - 1 \) to \( s.t \), i.e.,

\[
\forall v: V, s: Stream_V, t: \mathbb{Z} \cdot (\forall x: Proc \cdot \neg W_x(v.(s.t))) \Rightarrow (s.t - 1).v = (s.t).v
\]

The sum of the permissions of the processes for any variable \( v \) in any state of stream \( s \) is at most 1, i.e.,

\[
\forall v: V, s: Stream_V, t: \mathbb{Z} \cdot (\sum_{x \in Proc} \Pi_x(v.(s.t))) \leq 1
\]

Each variable in \( LVar_X \) may only be read or written to by the processes in \( X \), i.e.,

\[
\forall y: Proc\backslash X, v: LVar_X, s: Stream_V, t: \mathbb{Z} \cdot D_y.v.(s.t)
\]

These conditions essentially define the legal streams of the programs we consider. For the rest of this paper, we implicitly assume that the streams we consider are legal, i.e., satisfy healthiness conditions HC1-HC3.

Using these healthiness conditions, we obtain a number of relationships between the values of a variable and the permissions that a process has to access the variable. For example, we may prove if a process has read permission to a variable, then no process has write permission to the variable. Furthermore, if over an interval no process has write permission to a variable, then the variable must be stable over the interval.

**Lemma 4** Both of the following hold.

\[
\forall v: V, s: Stream_V, t: \mathbb{Z} \cdot (\exists x: Proc \cdot \mathcal{R}_x.v.(s.t)) \Rightarrow (\forall x: Proc \cdot \neg W_x(v.(s.t)) \quad (1)
\]

\[
\Downarrow (\forall x: Proc \cdot \neg W_x(v) \Rightarrow st.v \quad (2)
\]

The following result states that for any guard \( b \), if none of the processes write to \( v \in \text{vars}.b \) over an interval, then \( \oplus b \) holds (over the interval) iff \( \odot b \) holds, i.e., any apparent value of a state predicate is the only apparent value.

**Theorem 1** \( \Downarrow (\forall v: \text{vars}.b, x: Proc \cdot \neg W_x(v) \land \neg \text{Empty} \Rightarrow \oplus b = \odot b. \)

### 4 Interval-based semantics of parallel programs

#### 4.1 Chop and iteration

To formalise the semantics of compound commands, we define two operators on interval predicates: **chop**, which is used to formalise sequential composition, and **\( \omega \)-iteration**, which is used to formalise a possibly infinite iteration (e.g., a while loop).

The **chop** operator ‘;’ is a basic operator on two interval predicates [Mos00, DH12c], where \((p_1; p_2).\Delta\) holds iff either interval \( \Delta \) may be split into two parts so that \( p_1 \) holds in the first and \( p_2 \) holds in the second, or the least upper bound of \( \Delta \) is \( \infty \) and \( p_1 \) holds in \( \Delta \). Thus, we define

\[
(p_1; p_2).\Delta \equiv (\exists \Delta_1, \Delta_2: Inv \cdot (\Delta = \Delta_1 \cup \Delta_2) \land (\Delta_1 \prec \Delta_2) \land p_1.\Delta_1 \land p_2.\Delta_2) \lor (lub.\Delta = \infty \land p_1.\Delta)
\]
Note that $\Delta_1$ may be empty, in which case $\Delta_2 = \Delta$, and similarly $\Delta_2$ may empty, in which case $\Delta_1 = \Delta$. Furthermore, in the definition of chop, we allow the second disjunct $\text{lub}.\Delta = \infty \land \rho_1.\Delta$ to enable $\rho_1$ to model an infinite (divergent or non-terminating) program.

We define the possibly infinite iteration of an interval predicate $p$ as follows [Hay10], where the interval predicates are assumed to be ordered using universal implication `$\Rightarrow$' so that $false$ and $true$ form the top and bottom of the ordering, respectively.

$$p^\omega \equiv \mu z \cdot (p ; z) \lor \text{Empty}$$

Thus, $p^\omega$ is a least fixed point that defines either finite or infinite iteration of $p$ [BW99, Hay10]. We note that $(\text{Empty}; p) \equiv p$ and $p \equiv (p ; \text{Empty})$ hold for any interval predicate $p$.

### 4.2 States apparent to a process

Reasoning about the apparent states with respect to a process is not always adequate because it is not enough for an apparent state to exist; process $x$ must also be able to read the relevant variables this apparent state. Typically, it is not necessary for a process to be able to read all of the state variables to determine the apparent value of a given state predicate. In fact, in the presence of local variables (of other processes), it will be impossible for $x$ to read the value of each variable. Hence, we define a function $\text{apparent}_x.w$ where $w$ is the set of variables whose values $x$ needs to determine in order evaluate the given state predicate.

$$\text{apparent}_x.w,\Delta.s \equiv \{ \sigma; .\text{State}_w \mid \forall v: \exists r: \Delta \cdot (\sigma.v = s.t.v) \land R_x.v.(s.t)\}$$

Using this function, we are able to define state predicates definitely and possibly hold with respect to a process. For a state predicate $c$, interval $\Delta$ and stream $s$, we define:

$$(\otimes_x c).\Delta.s \equiv \forall \sigma: \text{apparent}_x.vars,c.\Delta.s \cdot c.\sigma$$

$$(\odot_x c).\Delta.s \equiv \exists \sigma: \text{apparent}_x.vars,c.\Delta.s \cdot c.\sigma$$

**Example 4** Suppose $(u, v, w) = (0, 0.42 \land R_x.u \land R_x.v \land \neg R_x.w).\langle s.1 \rangle$, $(u, v, w) = (1, 0.42 \land R_x.u \land R_x.v \land \neg R_x.w).\langle s.2 \rangle$ and $(u, v, w) = (1, 1.42 \land R_x.u \land R_x.v \land \neg R_x.w).\langle s.3 \rangle$ hold, where $s$ is a stream. Then $\odot_x(u < v).[1,3].s = true$ (which is the same scenario as Example 3). Note that the fact that $x$ cannot read $w$ in any of the states $s.1, s.2$ or $s.3$ is irrelevant with respect to evaluation of $u < v$. Suppose that $s'$ is another stream such that $s'.1, s'.2 = s.1, s.2$ and $(u, v, w) = (1, 1.42 \land R_x.u \land \neg R_x.v \land \neg R_x.w).\langle s'.3 \rangle$, i.e., the possible value of $v$ that $x$ can read is $0$. Hence, we have $\odot_x(u < v).[1,3].s' = false$ even though $\odot(u < v).[1,3].s = true$.

### 4.3 Interval-based command semantics

**Definition 2** For a state predicate $b$, variable $v$ and expression $e$, the abstract syntax of commands is given by $\text{Cmd}$ below, where $C, C_1, C_2 \in \text{Cmd}$.

$$\text{Cmd} \equiv \text{idle} \mid [b] \mid v := e \mid C_1 ; C_2 \mid C_1 \cap C_2 \mid C^\omega \mid C_1 || C_2$$

Thus, a command may either be an idle statement, a guard $[b]$, an assignment $v := e$, a sequential composition $(C_1 ; C_2)$, a non-deterministic choice $C_1 \cap C_2$, an iteration $C^\omega$, or a parallel composition $C_1 || C_2$. 

Proc. AVoCS 2012 8 / 18
Definition 3  The behaviour of a command $C$ given by the syntax in Definition 2 executed by a non-empty set of processes $X$ is given by interval predicate $\text{beh}_X.C$, which is defined inductively. For a process $x$, we let $\text{beh}_x$ denote $\text{beh}_{x_1}$.

\[
\begin{align*}
\text{beh}_{x,\text{Idle}} & \equiv \Box \forall v: \text{Var} \cdot \neg W_x.v \\
\text{beh}_{x,[b]} & \equiv (\Box_x b) \land (\forall v: \text{Var} \cdot \Box \neg W_x.v) \\
\text{beh}_{x,(v := e)} & \equiv \exists k: \text{Val} \cdot \text{beh}_{x,[e = k]}: \left( \Box \left( v = k \land W_x.v \right) \land \left( \forall u: \text{Var} \neg \left( v \land \neg W_x.u \right) \land \neg \text{Empty} \right) \right) \\
\text{beh}_X.(C_1; C_2) & \equiv \text{beh}_X.C_1 \land \text{beh}_X.C_2 \\
\text{beh}_X.(C_1 \cap C_2) & \equiv \text{beh}_X.C_1 \lor \text{beh}_X.C_2 \\
\text{beh}_X.C^\omega & \equiv (\text{beh}_X.C)^\omega \\
\text{beh}_X.(C_1 \parallel C_2) & \equiv \exists X_1, X_2 \cdot (X_1 \cup X_2 = X) \land (X_1 \cap X_2 = \emptyset) \land \text{beh}_{X_1}.(C_1; \text{Idle}) \land \text{beh}_{X_2}.(C_2; \text{Idle})
\end{align*}
\]

We interpret the behaviours of these commands as follows. By (3), an Idle command executed by process $x$ guarantees that $x$ does not modify any of the variables of the system. Note that a stronger specification of the behaviour of Idle is $\Box \forall v: \text{Var} \cdot D_x.v$, which guarantees that process $x$ is denied access to $v$. However, we choose to define as weak a specification as possible, which allows an implementation of Idle to read from variable $v$. By (4), the behaviour of a guard evaluation states that there must be an apparent state in which $b$ holds and process $x$ has permission to read each of the variables of $b$ in this apparent state. Furthermore, process $x$ does not write to any of the system variables over the interval in which the evaluation occurs. By (5), an assignment $v := e$ evaluates the expression $e$ over an interval to a value, say $k$, then updates the value of $v$ to $k$. Process $x$ does not have read or write permission to any of the other variables while $v$ is being updated. For any state predicate $c$, because $\Box c$ holds trivially in an empty interval, we must explicitly include conjunct $\neg \text{Empty}$ to ensure that the value of $v$ is updated. We note that the value of any shared variable within $e$ may be changing over the evaluation interval, i.e., during the execution of $[e = k]$, and hence the value assigned to $v$ may be non-deterministic. By (6), the behaviour of the sequential composition is defined using chop. We note that the chop allows $C_1$ to execute forever, e.g., if $C_1$ models an infinite loop. By (7), the behaviour of the non-deterministic choice is the behaviour of either $C_1$ or $C_2$. By (8), the behaviour of an iteration $C^\omega$ is the behaviour of $C$ iterated a potentially infinite number of times. By (9), the parallel composition behaves as both $C_1$ and $C_2$, but we must execute an Idle command after both $C_1$ and $C_2$ to achieve schedulability of the two programs.

We assume the existence of commands False and True whose behaviours for a non-empty set of processes $X$ are defined as follows:

\[
\begin{align*}
\text{beh}_X.\text{False} & \equiv \text{false} \\
\text{beh}_X.\text{True} & \equiv \text{true}
\end{align*}
\]

Thus, False is an infeasible command that has no behaviours and True is a chaotic command that allows any behaviour.
4.4 Refinement, equivalence and enforced conditions

Refinement of commands is defined using behavioural implication. That is, command $C$ refines $A$ with respect to a set of processes $X$ iff every behaviour of $C$ is a possible behaviour of $A$.

**Definition 4** Given commands $A$ and $C$ and a set of processes $X$, we say $A$ is refined by $C$ with respect to $X$ (denoted $A \sqsubseteq_X C$) iff $\text{beh}_X.C \Rightarrow \text{beh}_X.A$ holds.

We write $A \sqsubseteq \sqsupseteq_X C$ if both $A \sqsubseteq_X C$ and $C \sqsubseteq_X A$ hold. Note that $\sqsubseteq_X$ is a pre-order, i.e., is a reflexive and transitive relation. Furthermore, for any command $C$, both $C \sqsubseteq_X \text{False}$ and $\text{True} \sqsubseteq_X C$ hold trivially.

A convenient approach to constraining the behaviour of a command is to use enforced conditions, which have been used to derive concurrent programs [Don09, DH12b]. Provided that $C$ is a command and $d$ is an interval predicate, $C$ with an enforced condition $d$ (denoted $\text{Enf } d \cdot C$) is a command that behaves as $C$ and in addition ensures that $d$ holds [Don09, DH12b]. In particular, the behaviour of $\text{Enf } d \cdot C$ is defined as follows.

$$\text{beh}_X.(\text{Enf } d \cdot C) \equiv d \land \text{beh}_X.C$$

Note if $\neg d$ holds, then $\text{Enf } d \cdot C$ has no behaviours.

The lemma below states that introducing an enforced property and strengthening an existing property results in a refinement.

**Lemma 5** Both of the following hold.

$$C \sqsubseteq_X \text{Enf } d \cdot C$$

$$d' \Rightarrow d \Rightarrow \text{Enf } d \cdot C \sqsubseteq_X \text{Enf } d' \cdot C$$

The proofs of both properties above are straightforward by expanding the definitions. Enforced properties may also be used to specify behaviours of other processes, i.e., the environment of a process. The lemma below states that given there is no interference to the variables of $b$ within the interval in which $b$ is evaluated, the behaviour of the guard evaluation implies $\oplus b$ and hence every apparent state satisfies $b$.

**Lemma 6** For any state predicate $b$ and process $x$,

$$\text{beh}_x.(\text{Enf } \exists !v: \text{vars}.b \cdot \neg I_{\{x\}}.v) \Rightarrow \oplus_x b$$

4.5 Example behaviours

We expand the behaviours of two simple programs to illustrate how interval-based behaviours together with non-deterministic evaluation and fractional permissions accurately captures fine-grained interleaving of parallel processes. In particular, in Section 4.5.2 we return to the program from Example 1 to demonstrate how our framework allows apparent states to be observed.

---

4 It is possible to obtain a number of refinement laws on behaviours of commands, but we do not discuss these here.

5 These have similar properties to coercions in the refinement calculus.
4.5.1 Parallel assignments

Suppose \( v \in LVar_{\{x,y\}} \) and that the initial value of \( v \) is 0. We consider the behaviour of commands 
\( v := v + 1 \) and \( v := v + 10 \) executed by processes \( x \) and \( y \), respectively, over a finite interval.

\[
beh_{x,y}((v := v + 1) \land (v := v + 10)) \\
= \exists \kappa_x, \kappa_y \cdot (beh_{x,[\kappa_x = v + 1]} \land \Box(v = \kappa_x \land \neg W_x.v \land \neg Empty) \land \Box(W_x.v \land \neg Empty)) \\
\quad \land \Box(W_y.v) \land \Box(W_y.v)
\]

To simplify the interval predicate above, we make the following substitutions.

\[
x_1 \equiv beh_x,[\kappa_x = v + 1] \\
x_2 \equiv \Box(v = \kappa_x \land \neg W_x.v \land \neg Empty) \\
x_3 \equiv \Box \neg W_x.v
\]

Hence, we obtain:

\[
\exists \kappa_x, \kappa_y \cdot (x_1 \land x_2 \land x_3) \land (y_1 \land y_2 \land y_3)
\]

using the write permission to order behaviours

\[
\exists \kappa_x, \kappa_y \cdot ((x_1 \land (y_1 \land y_2 \land y_3)); (x_2 \land y_3); (x_3 \land y_3)) \lor ((y_1 \land (x_1 \land x_2 \land x_3)); (x_3 \land y_2); (x_3 \land y_3))
\]

Let us consider the first disjunct in more detail.

First, we note that the disjunct already captures behaviours such as

\[
(x_1 \land (y_1 \land y_2)); (x_2 \land y_3); (x_3 \land y_3)
\]

i.e., the write to \( v \) by process \( y \) may occur immediately before the write by process \( x \), i.e., without any gaps. This is because because \( \Box c \) holds (for any state predicate \( c \)) in an empty interval.

Second, we note that the guard evaluation \( beh_{x,[\kappa_x = v + 1]} \) in process \( x \) (i.e., \( x_1 \)) and the assignment \( v \) in process \( y \) (i.e., \( y_1, y_2, y_3 \)) do not conflict, even though \( x_1 \) and \( y_2 \) conflict, because the guard evaluation only requires that there is some state in which \( v \) can be read — process \( x \) may read \( v \) during execution of \( y_1 \) or \( y_3 \) because, both \( y_1 \) and \( y_3 \) guarantee \( \Box \neg W_y.v \). Hence it is possible for process \( x \) to read the value of \( v \) before or after process \( y \) updates \( v \).

By Lemma 6, using the stability of \( v \in LVar_{\{x,y\}} \) during guard evaluation and because the initial value of \( v \) is assumed to be 0, the first disjunct above simplifies to:

\[
\exists \kappa_x \cdot \left( x_1 \land \Box(v = 0 \land \neg W_y.v) \lor \Box(v = 10 \land \neg W_y.v) \lor \Box(v = 10 \land \neg W_y.v) \land \neg Empty \right)
\]

Process \( x \) may read variable \( v \) (for the guard evaluation \( x_1 \)) either before or after process \( y \) writes to \( v \). Hence, we obtain the following:

\[
(\Box(v = 0)); \Box(v = 10); \Box(v = 1)) \lor (\Box(v = 0); \Box(v = 10); \Box(v = 11))
\]

Similarly expanding the second disjunct above results in the following behaviour:
\[(\mathbb{E}(v = 0); \mathbb{E}(v = 1); \mathbb{E}(v = 10)) \lor (\mathbb{E}(v = 0); \mathbb{E}(v = 1); \mathbb{E}(v = 11))\]

Thus, by allowing execution of \texttt{Idle} after each assignment as part of the parallel composition, we obtain the expected behaviour of the two parallel assignments to \(v\). Note that by using an interval-based logic, we are not only able to specify the possible values of \(v\), but also the sequence of values of \(v\).

### 4.5.2 Guard evaluation with parallel assignments

We now return to the original problem described in Example 1 and consider the evaluation of the guard \(u < v\) in parallel with the two assignments. We assume \(u, v \in LVar_{\{x, y\}}\) to prevent processes different from \(x\) and \(y\) from modifying \(u\) and \(v\). Hence, we have:

\[
\mathit{beh}_{\{x, y\}}[[u < v]]((u := 1; v := 1))
\]

\[\Rightarrow\] expanding definitions, using stability properties

\[
\Box_x(u < v) \land \mathbb{E}(\neg W_x.u \land \neg W_x.v) \land \left\{\begin{array}{ll}
(\mathbb{E}(u, v = 0, 0) \land \neg W_y.u \land \neg W_y.v); \\
(\mathbb{E}(u, v = 1, 0) \land (\neg W_y.u; \mathbb{E} \neg W_y.u)); \\
(\mathbb{E}(u, v = 1, 1) \land (\mathbb{E} \neg W_y.v; \mathbb{E} \neg W_y.v))
\end{array}\right.
\]

Hence, we obtain the scenario described in Example 3, and it is possible for the guard \(u < v\) to evaluate to \texttt{true} in process \(x\) even though there is no actual state in which \(u < v\) holds.

### 5 Compositional reasoning

To accommodate scalable proofs, we develop rely/guarantee-style proof methods where the rely condition is used to describe properties of the environment of a program. Previous methods for rely/guarantee reasoning, including those over intervals assume an interleaving between environment and component transitions [Jon83, STER11]. As we have seen with our sets of apparent states evaluators, such models fail to observe states in which several predicates involve multiple non-stable variables. To address this, Coleman and Jones use a small-step operational semantics that allows rely conditions to formalise the pre-post states after partially evaluating an expression [CJ07]. Jones and Pierce introduce some new notation for determining the set of all “possible values” that variable \(v\) may have during the execution of an operation [JP11].

In this paper, we have used an interval-based approach in which a program and its environment execute in a truly concurrent manner. Defining rely/guarantee rules in this model are made difficult because one must synchronise access to shared variables to avoid conflicting behaviour. In a real-time setting (with continuous variables), a solution to this problem is to distinguish between actual values and the rate of change of a variable [DH12d, DH12c], where a real-time controller is able to modify the rate of change of a variable and the environment modifies the value of the variable based on its rate of change. Such an approach does not work for the discrete model used in this paper. However, it turns out that the permission-based approach we have used in this paper addresses the synchronisation problem for discrete systems by allowing one to distinguish streams with conflicting variable accesses.
For an interval predicate $r$, and command $C$, we let $\text{RELY} \cdot C$ denote a command with a rely condition $r$, where

$$\text{beh}_X(\text{RELY} \cdot C) \equiv r \Rightarrow \text{beh}_X.C$$

Hence, $\text{RELY} \cdot C$ consists of an execution of $C$ under the assumption that $r$ holds [Jon83, JP11, CJ07]. Note that if $\neg r$ holds, then the behaviour of $\text{RELY} \cdot C$ is chaotic, i.e., any behaviour is allowed. The following lemma allows one to refine programs that execute under a rely conditions.

**Lemma 7** Each of the following holds.

\begin{align*}
    r \Rightarrow r' & \Rightarrow \text{RELY} \cdot C \subseteq X \text{RELY} \cdot C & (12) \\
    r \land \text{beh}_X.C & \Rightarrow \text{beh}_X.A \Rightarrow \text{RELY} \cdot A \subseteq X \text{RELY} \cdot C & (13) \\
    r \land \text{beh}_X.C & \Rightarrow d \Rightarrow \text{RELY} \cdot \text{ENF} \cdot d \cdot C \subseteq X \text{RELY} \cdot C & (14) \\
    A \subseteq X \text{ENF} \cdot C & \Rightarrow \text{RELY} \cdot A \subseteq X C & (15)
\end{align*}

We use the following theorem to decompose a proof that a parallel composition $A_1 \parallel A_2$ in the context of a rely condition $r$ is refined by $C_1 \parallel C_2$. Within Theorem 2, $r$ is an overall rely condition on the parallel composition $A_1 \parallel A_2$, $r_1$ is a rely condition for $A_1$ and $r_2$ a rely condition on $A_2$.

**Theorem 2** $(\text{RELY} \cdot A_1 \parallel A_2) \subseteq X (C_1 \parallel C_2)$ holds if there exist $X_1, X_2 \subseteq X$ where $X = X_1 \cup X_2$ and $X_1 \cap X_2 = \emptyset$ and rely conditions $r_1$ and $r_2$, such that both of the following hold.

\begin{align*}
    (\text{RELY} \land r_1 \cdot A_1) \subseteq X_1 C_1 & \land (\text{RELY} \land r_2 \cdot A_2) \subseteq X_2 C_2 & (16) \\
    (r \land \text{beh}_{X_2}.C_2 \Rightarrow r_1) & \land (r \land \text{beh}_{X_1}.C_1 \Rightarrow r_2) & (17)
\end{align*}

We may decompose a rely condition over the whole interval into its subintervals if the rely condition splits [Hay08, DH12d]. We say an interval predicate $p$ splits iff for any interval $\Delta$, if $p.\Delta$ holds, then for any subinterval $\Delta' \subseteq \Delta$, $p.\Delta'$ holds.

**Theorem 3** If $r$ splits, then each of the following holds.

\begin{align*}
    (\text{RELY} \cdot C_1; C_2) \subseteq X & (\text{RELY} \cdot C_1); (\text{RELY} \cdot C_2) & (18) \\
    \text{RELY} \cdot C^\omega \subseteq X & (\text{RELY} \cdot C)^\omega & (19) \\
    (\text{RELY} \cdot A \subseteq X C) & \Rightarrow (\text{RELY} \cdot A^\omega \subseteq X C^\omega) & (20)
\end{align*}

### 6 Conclusions and future work

We have presented an interval-based approach to modelling the behaviour of concurrent programs, which better represents the “real-world” behaviour in multicore/multiprocessor systems. Conflicting accesses to shared variables are modelled using Boyland’s fractional permissions and non-deterministic expression evaluators are used to reason about apparent behaviour. Using rely conditions, we have developed rules for decomposing proofs of systems that involve multiple parallel components, both over the parallel and sequential compositions.
The underlying semantics of Interval Temporal Logic that we present differs from Moszkowski [Mos00]. In particular, we consider complete streams (the behaviour over all discrete times) and the behaviour over an interval is given by an interval predicate, which restricts the view of the complete stream to the interval under consideration. An advantage of our model is that it allows properties outside the given interval to be given in a straightforward manner [DH12a, DH12b, DH12c, DH12d]. Furthermore, it is possible to extend this theory to reason about continuous behaviour [DH12b, DH12c, DH12d]. A second difference with [Mos00] is that Moszkowski allows adjoining intervals to overlap at their boundary, whereas we require that adjoining intervals are disjoint. Adjoining intervals that overlap at the boundary are problematic when using fractional permissions if a process writes to a variable, then immediately performs a guard evaluation that reads from the same variable. If there is an overlap, the write permission for the update conflicts (which must ensure write permission on the variable) conflicts with guard evaluation at the boundary (which must ensure no write permission is held).

Our treatments of interval predicates has an algebraic structure [BW99] that we hope to further exploit in our reasoning. For example, rule (15) is similar to the refinement calculus rule \((\{b\} A \sqsubseteq C) \Leftrightarrow (A \sqsubseteq \{b\} C)\) that allows an assertion \(\{b\}\) to be turned into a coercion \(\{b\}\) (and vice versa) [BW99]. We aim to develop rules that would allow one to exploit the abstract algebraic properties in our proofs.

Acknowledgements. We thank Lindsay Groves for helpful comments on an earlier draft, including the suggestion that we consider a permission-based approach.

Bibliography


A Proofs of theorems

Theorem 1 \( \exists (\forall: \text{vars}, b, x: \text{Proc} \cdot \neg W_x, v) \land \neg \text{Empty} \Rightarrow \odot b = \odot b. \)

Proof. The proof for \( \odot b \land \neg \text{Empty} \Rightarrow \odot b \) follows from the definitions of \( \odot \) and \( \odot \). For the other direction, we have the following calculation.

\[
\begin{align*}
\exists (\forall: \text{vars}, b, x: \text{Proc} \cdot \neg W_x, v) & \Rightarrow (\odot b \Rightarrow \odot b) \\
\Leftrightarrow & \quad \text{logic} \\
\exists (\forall: \text{vars}, b, x: \text{Proc} \cdot \neg W_x, v) \land \odot b & \Rightarrow \odot b \\
\Leftrightarrow & \quad (2) \\
st.(\text{vars}, b) \land \odot b & \Rightarrow \odot b \\
\Leftrightarrow & \quad \text{Lemma 3} \\
& \quad \text{true} \quad \Box
\end{align*}
\]

Theorem 2 \( (\text{Relv} r \cdot A_1 \parallel A_2) \subseteq_X C_1 \parallel C_2 \) holds if there exist \( X_1, X_2 \subseteq X \) where \( X = X_1 \cup X_2 \) and \( X_1 \cap X_2 = \emptyset \) and rely conditions \( r_1 \) and \( r_2 \), such that both of the following hold.

\[
\begin{align}
(\text{Relv} r \land r_1 \cdot A_1) & \subseteq_{X_1} C_1 \land (\text{Relv} r \land r_2 \cdot A_2) \subseteq_{X_2} C_2 \quad (21) \\
(\text{Relv} r \land \text{beh}_{X_2} C_2) & \Rightarrow r_1 \land (\text{Relv} r \land \text{beh}_{X_1} C_1 \Rightarrow r_2) \quad (22)
\end{align}
\]

Proof.\( \text{ (21)} \)

\[
= \quad \text{definition of } \subseteq_X, \text{logic } (a \Rightarrow (b \Rightarrow c)) = (a \land b \Rightarrow c) \\
(r \land r_1 \land \text{beh}_{X_1} C_1 \Rightarrow \text{beh}_{X_1} A_1) \land (r \land r_2 \land \text{beh}_{X_2} C_2 \Rightarrow \text{beh}_{X_2} A_2) \\
\Rightarrow \quad \text{logic} \\
\Rightarrow \quad (r \land r_1 \land \text{beh}_{X_1} C_1 \land r_2 \land \text{beh}_{X_2} C_2 \Rightarrow \text{beh}_{X_1} A_1 \land \text{beh}_{X_2} A_2) \quad (22) \\
\Rightarrow \quad r \land \text{beh}_{X_1} C_1 \land \text{beh}_{X_2} C_2 \Rightarrow \text{beh}_{X_1} A_1 \land \text{beh}_{X_2} A_2
\]

Hence we have:

\[
\exists X_1, X_2 \cdot (X_1 \cup X_2 = X) \land (X_1 \cap X_2 = \emptyset) \land (21) \\
\Rightarrow \quad \text{calculation above and logic} \\
\exists X_1, X_2 \cdot (X_1 \cup X_2 = X) \land (X_1 \cap X_2 = \emptyset) \land r \land \text{beh}_{X_1} C_1 \land \text{beh}_{X_2} C_2 \Rightarrow \text{beh}_{X_1} A_1 \land \text{beh}_{X_2} A_2 \\
= \quad \text{definitions} \\
(\text{Relv} r \cdot A_1 \parallel A_2) \subseteq_X C_1 \parallel C_2 \quad \Box
\]

Theorem 3 If \( r \) splits, then each of the following holds.

\[
\begin{align}
(\text{Relv} r \cdot C_1 : C_2) & \subseteq_X (\text{Relv} r \cdot C_1) : (\text{Relv} r \cdot C_2) \quad (23) \\
\text{Relv} r \cdot C^o \subseteq_X (\text{Relv} r \cdot C)^o \quad (24) \\
(\text{Relv} r \cdot A \subseteq_X C) & \Rightarrow (\text{Relv} r \cdot A^o \subseteq_X C^o) \quad (25)
\end{align}
\]

Proof of (23).

\[
\text{beh}_X.(\text{Relv} r \cdot C_1 : C_2) \\
\quad \equiv \quad \text{expanding definitions}
\]
(r ⇒ (behX.C1; behX.C2))
⇔ r splits
r ⇒ ((r ⇒ behX.C1); (r ⇒ behX.C2))
⇔ logic
(r ⇒ behX.C1); (r ⇒ behX.C2)
≡ definitions
behX.(Rely r · C1); (Rely r · C2)
□

Proof of (24).

Rely r · Cω ⊑X (Rely r · C)ω
= (15) of Lemma 7
Cω ⊑X Enr · (Rely r · C)ω
= Cω ⊑X (C; Cω) ∨ Empty by ω induction
C; (Enr · (Rely r · C)ω) ∨ Empty ⊑X Enr · (Rely r · C)ω
= (15) of Lemma 7
Rely r · (C; (Enr · (Rely r · C)ω) ∨ Empty) ⊑X (Rely r · C)ω

We obtain the following series of refinements to establish the condition above.

Rely r · (C; (Enr · (Rely r · C)ω) ∨ Empty)
≤X distribute rely
(Rely r · C; (Enr · (Rely r · C)ω)) ∨ (Rely r · Empty)
≤X r splits, (23) and Rely r · C ⊑X C
((Rely r · C); (Rely r · Enr · (Rely r · C)ω)) ∨ Empty
≤X (14)
((Rely r · C); (Rely r · (Rely r · C)ω)) ∨ Empty
≤X Rely r · C ⊑X C
((Rely r · C); (Rely r · C)ω) ∨ Empty
≤X ω folding
(Rely r · C)ω
□

Proof of (25).

Rely r · Aω
≤X (24), r splits
(Rely r · A)ω
≤X assumption Rely r · A ⊑X C
Cω
□