Electronic Communications of the EASST
Volume 53 (2012)

Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)

Fractional Permissions and Non-Deterministic Evaluators in Interval Temporal Logic

Brijesh Dongol, John Derrick, Ian J. Hayes

15 pages

Guest Editors: Gerald Lüttgen, Stephan Merz
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

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

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 of 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 shared variables from the pre-state of a transition because variables may be modified

* Brijesh Dongol and John Derrick are sponsored by EPSRC Grant EP/J003727/1.
† Ian J. Hayes is sponsored by ARC Discovery Grant DP0987452.

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** S_1 **else** S_2 **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 S_2 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 S_1.*

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 S_1 (see Section 4.5.2).

An advantage of using Interval Temporal Logic is that the behaviour of the parallel composition 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 components 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 Tempo-

ral 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 $\mathbb{Z}$), and hence the set of all intervals is:

$$Intv \;\widehat{=}\; \{\Delta \subseteq \mathbb{Z} \mid \forall t,t' : \Delta \bullet \forall u : \mathbb{Z} \bullet t \leq u \leq t' \Rightarrow u \in \Delta\}$$

Using '.' for function application, we let $\mathsf{lub}.\Delta$ and $\mathsf{glb}.\Delta$ denote the *least upper* and *greatest lower* bounds of an interval Δ, respectively, and we define $\mathsf{lub}.\varnothing \mathrel{\widehat{=}} -\infty$ and $\mathsf{glb}.\varnothing \mathrel{\widehat{=}} \infty$. If the size of Δ is infinite and $\mathsf{glb}.\Delta \in \mathbb{Z}$, then $\mathsf{lub}.\Delta = \infty$ (i.e,. is not a member of $\mathbb{Z}$) and if Δ is infinite and $\mathsf{lub}.\Delta \in \mathbb{Z}$ then $\mathsf{glb}.\Delta = -\infty$.

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

$$\Delta \propto \Delta' \;\widehat{=}\; \Delta \neq \varnothing \wedge \Delta' \neq \varnothing \Rightarrow (\mathsf{lub}.\Delta < \mathsf{glb}.\Delta') \wedge (\Delta \cup \Delta' \in Intv)$$

Thus, $\Delta \propto \Delta'$ holds if and only if both $\mathsf{lub}.\Delta < \mathsf{glb}.\Delta'$ holds (which ensures Δ' follows Δ and that Δ is disjoint with Δ') and $\Delta \cup \Delta' \in Intv$ holds (which ensures that the union of Δ and Δ' is contiguous). Note that both $\Delta \propto \varnothing$ and $\varnothing \propto \Delta$ hold trivially.

We use $\mathsf{Empty}.\Delta$ to denote that the given interval Δ is empty.

$$\mathsf{Empty}.\Delta \;\widehat{=}\; \Delta = \varnothing$$

Given that variable names are taken from the set *Var*, a *state space* over a set of variables $V \subseteq Var$ is given by $State_V \mathrel{\widehat{=}} V \to Val$ and a *state* is a member of $State_V$, i.e., a state is a total function mapping variables in V to values in *Val*. Note that we do not distinguish between variables and the 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 $Stream_V \mathrel{\widehat{=}} \mathbb{Z} \to State_V$, which maps each time in $\mathbb{Z}$ to a state over V. A *predicate* over type T is a total function $\mathcal{P}T \mathrel{\widehat{=}} T \to \mathbb{B}$ mapping each member of T to a Boolean. For example $\mathcal{P}State_V$ and $\mathcal{P}Stream_V$ denote state and stream predicates, respectively. To facilitate reasoning about specific parts of a stream, we use *interval predicates*, which have type $IntvPred_V \mathrel{\widehat{=}} Intv \to \mathcal{P}Stream_V$.

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, Δ is an interval and s is a stream, we have $(p_1 \wedge p_2).\Delta.s = (p_1.\Delta.s \wedge p_2.\Delta.s)$. When reasoning about properties of programs, we would like to state that whenever a property p_1 holds over any interval Δ and stream s, a property p_2 also holds over Δ and s. Hence, we define universal implication for $p_1, p_2 \in IntvPred_V$ as

$$p_1 \Rrightarrow p_2 \;\widehat{=}\; \forall \Delta : Intv, s : Stream_V \bullet p_1.\Delta.s \Rightarrow p_2.\Delta.s$$

We say $p_1 \equiv p_2$ holds iff both $p_1 \Rrightarrow p_2$ and $p_2 \Rrightarrow p_1$ 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 that a state predicate containing multiple variables 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 Δ and stream s are defined below:

$$\begin{aligned} \mathit{states}.\Delta.s &\mathrel{\widehat{=}} \{\sigma{:}\,\mathit{State}_V \mid \exists t{:}\,\Delta \bullet \sigma = s.t\} \\ \mathit{apparent}.\Delta.s &\mathrel{\widehat{=}} \{\sigma{:}\,\mathit{State}_V \mid \forall v{:}\,V \bullet \exists t{:}\,\Delta \bullet \sigma.v = s.t.v\} \end{aligned}$$

Example 2 (Sets of states) *Consider the statements $u := 1\,;\ v := 1$ from Example 1 which we assume are executed over an interval Δ 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:*

$$\mathit{states}.\Delta.s \quad = \quad \Big\{\{u \mapsto 0, v \mapsto 0\}, \{u \mapsto 1, v \mapsto 0\}, \{u \mapsto 1, v \mapsto 1\}\Big\}\,.$$

The sets-of-states approach evaluates the given state predicate in one of the actual states of the observation interval. For example, $u \le v$ evaluated in s within Δ above has possible values $\{\mathit{false}, \mathit{true}\}$. 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 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]*:*

$$\mathit{apparent}.\Delta.s \quad = \quad \Big\{\{u \mapsto 0, v \mapsto 0\}, \{u \mapsto 1, v \mapsto 1\}, \{u \mapsto 0, v \mapsto 1\}, \{u \mapsto 1, v \mapsto 0\}\Big\}.$$

Note that $\mathit{apparent}.\Delta.s = \mathit{states}.\Delta.s \cup \{\{u \mapsto 0, v \mapsto 1\}\}$. Using a sets of apparent states evaluation, a predicate is evaluated in one of the states of $\mathit{apparent}.\Delta.s$. Hence, unlike the sets of states approach, the possible values of both $u \le v$ and $u < v$ are $\{\mathit{false}, \mathit{true}\}$. 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 $\boxdot c$ and $\diamondsuit c$, 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 $\circledast c$) and c *possibly* holds (denoted $\circledcirc c$) in all apparent states of a stream within an interval.

$$(\boxdot c).\Delta.s \;\widehat{=}\; \forall \sigma : states.\Delta.s \bullet c.\sigma \qquad (\circledast c).\Delta.s \;\widehat{=}\; \forall \sigma : apparent.\Delta.s \bullet c.\sigma$$
$$(\diamondsuit c).\Delta.s \;\widehat{=}\; \exists \sigma : states.\Delta.s \bullet c.\sigma \qquad (\circledcirc c).\Delta.s \;\widehat{=}\; \exists \sigma : apparent.\Delta.s \bullet c.\sigma$$

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

Lemma 1 *Both $\circledast c \Rrightarrow \boxdot c$ and $\diamondsuit c \Rrightarrow \circledcirc c$ hold, but the converses of both implications are not necessarily true.*

We say a variable v is *stable* at time t in stream s (denoted $\mathsf{stable\_at}.v.t.s$) iff the value of v in s at time t does not change from its value at time $t-1$, i.e.,

$$\mathsf{stable\_at}.v.t.s \;\widehat{=}\; s.t.v = s.(t-1).v$$

Variable v is stable over an interval Δ in a stream s (denoted $\mathsf{stable}.v.\Delta.s$) iff the value of v is stable at each time within Δ. A set of variables V is *stable* in Δ (denoted $\mathsf{stable}.V.\Delta$) iff each variable in V is stable in Δ. Thus, we define:

$$\mathsf{stable}.v.\Delta.s \;\widehat{=}\; \forall t : \Delta \bullet \mathsf{stable\_at}.v.t.s \qquad \mathsf{stable}.V.\Delta \;\widehat{=}\; \forall v : V \bullet \mathsf{stable}.v.\Delta$$

Note that every variable is stable in an empty interval and the empty set of variables is stable in any interval, i.e., both $\mathsf{stable}.V.\varnothing$ and $\mathsf{stable}.\varnothing.\Delta$ hold trivially.

We let $vars.c$ 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 Δ, then c definitely holds in Δ iff c always holds in Δ and c possibly holds in Δ iff c holds sometime in Δ [HBDJ11].

Lemma 2 *For a state predicate c and variable v,*
$\mathsf{stable}.(vars.c \backslash \{v\}) \Rrightarrow (\circledast c = \boxdot c) \wedge (\circledcirc c = \diamondsuit c)$.

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 a state predicate c,*
$\mathsf{stable}.(vars.c) \wedge \neg \mathsf{Empty} \Rrightarrow (\circledast c = \boxdot c = \circledcirc c = \diamondsuit c)$.

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 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 between 0 and 1. A process has write-only access to a variable v 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 *Proc* denote the set of all process identifiers and assume that every state contains a *permission* variable Π whose value in state $\sigma \in State_V$ is a function of type

$$V \to Proc \to \{n : \mathbb{Q} \mid 0 \leq n \leq 1\}$$

Note that it is possible for permissions to be distributed differently within states σ, σ' even if the values of the standard variables in σ and σ' are identical, i.e., it is possible to get $\sigma.\Pi \neq \sigma.\Pi'$ even if $(\{\Pi\} \lhd \sigma) = (\{\Pi\} \lhd \sigma')$ holds, where '$\lhd$' denotes the domain anti-restriction.

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

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.

$$\mathcal{R}.v.x \;\widehat{=}\; 0 < \Pi.v.x < 1 \qquad \mathcal{W}.v.x \;\widehat{=}\; \Pi.v.x = 1 \qquad \mathcal{D}.v.x \;\widehat{=}\; \Pi.v.x = 0$$

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 $\mathcal{W}.v.x$ and $\mathcal{R}.v.x$, respectively. Thus, $\mathcal{W}.v.x$ does not grant process x permission to write to v in the transition from $s.t$ to $s.(t+1)$ (and similarly $\mathcal{R}.v.x$).

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 different from x.

$$\mathcal{I}.v.x \;\widehat{=}\; \exists y : Proc \setminus \{x\} \bullet \mathcal{W}.v.y$$

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.

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 $LVar_X \subseteq Vars$.

HC1 If no process has write access to v within an interval, then the value of v does not change within the interval, i.e.,

$$\boxdot(\forall x{:}\,Proc \bullet \neg\mathcal{W}.v.x) \quad \Rightarrow \quad \mathsf{stable}.v$$

HC2 The sum of the permissions of the processes on any variable v is at most 1, i.e.,

$$\boxdot((\Sigma_{x \in Proc}\Pi.v.x) \quad \leq \quad 1)$$

HC3 Each local variable in $v \in LVar_X$ may only be read or written to by the processes in X, i.e.,

$$\boxdot(\forall y{:}\,Proc \backslash X \bullet \mathcal{D}.v.y)$$

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**. Such healthiness conditions can be explicitly added to the rely conditions of a program.

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 that 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 for any variable v*

$$\boxdot((\exists x{:}\,Proc \bullet \mathcal{R}.v.x) \quad \Rightarrow \quad (\forall x{:}\,Proc \bullet \neg\mathcal{W}.v.x)) \tag{1}$$

$$\boxdot(\forall x{:}\,Proc \bullet \neg\mathcal{W}.v.x) \quad \Rrightarrow \quad \mathsf{stable}.v \tag{2}$$

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

Theorem 1 $\boxdot(\forall v{:}\,vars.b, x{:}\,Proc \bullet \neg\mathcal{W}.v.x) \wedge \neg\mathsf{Empty} \Rrightarrow (\circledast b = \circledcirc b)$.

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 *ω-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, DH12b], where $(p_1 \,;\, p_2).\Delta$ holds iff either interval Δ 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 Δ is ∞ and p_1 holds in Δ. Thus, we define

$$(p_1 \,;\, p_2).\Delta \quad \widehat{=} \quad \begin{array}{l}\Big(\exists \Delta_1, \Delta_2{:}\,Intv \bullet (\Delta = \Delta_1 \cup \Delta_2) \wedge (\Delta_1 \propto \Delta_2) \wedge p_1.\Delta_1 \wedge p_2.\Delta_2\Big) \vee \\ (\mathsf{lub}.\Delta = \infty \wedge p_1.\Delta)\end{array}$$

Note that Δ_1 may be empty, in which case $\Delta_2 = \Delta$, and similarly Δ_2 may empty, in which case $\Delta_1 = \Delta$. Furthermore, in the definition of chop, we allow the second disjunct $\mathsf{lub}.\Delta = \infty \wedge p_1.\Delta$ to enable p_1 to model an infinite (divergent or non-terminating) program. We note that for any interval predicate p, both $(\mathsf{Empty}\,;\, p) \equiv p$ and $p \equiv (p\,;\, \mathsf{Empty})$ hold.

We define the possibly infinite iteration of an interval predicate p as follows [DHMS12], where the interval predicates are assumed to be ordered using universal implication '$\Rrightarrow$' so that *false* and *true* form the bottom and top of the ordering, respectively.

$$p^{\omega} \;\widehat{=}\; \nu z \bullet (p\,;\, z) \vee \mathsf{Empty}$$

Thus, p^{ω} is the greatest fixed point that defines either finite or infinite iteration of p [DHMS12].

4.2 States apparent to a process

Reasoning about the apparent states with respect to a process x using function *apparent* 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 in 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 $apparent_{x,W}$ where W is the set of variables whose values process x needs to determine in order evaluate the given state predicate.

$$apparent_{x,W}.\Delta.s \;\widehat{=}\; \{\sigma{:}\,State_W \mid \forall v{:}\,W \bullet \exists t{:}\,\Delta \bullet (\sigma.v = s.t.v) \wedge \mathcal{R}.v.x.(s.t)\}$$

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 Δ and stream s, we define:

$$\begin{aligned}(\circledast_x\, c).\Delta.s &\;\widehat{=}\; \forall \sigma{:}\,apparent_{x,vars.c}.\Delta.s \bullet c.\sigma\\ (\circledcirc_x\, c).\Delta.s &\;\widehat{=}\; \exists \sigma{:}\,apparent_{x,vars.c}.\Delta.s \bullet c.\sigma\end{aligned}$$

Example 4 *Suppose* $(u,v,w = 0,0,42 \wedge \mathcal{R}.u.x \wedge \mathcal{R}.v.x \wedge \neg\mathcal{R}.w.x).(s.1)$, $(u,v,w = 1,0,42 \wedge \mathcal{R}.u.x \wedge \mathcal{R}.v.x \wedge \neg\mathcal{R}.w.x).(s.2)$ *and* $(u,v,w = 1,1,42 \wedge \mathcal{R}.u.x \wedge \mathcal{R}.v.x \wedge \neg\mathcal{R}.w.x).(s.3)$ *hold, where* s *is a stream. Then* $\circledcirc_x(u < v).[1,3].s = true$ *(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* $(u,v,w = 1,1,42 \wedge \mathcal{R}.u.x \wedge \neg\mathcal{R}.v.x \wedge \neg\mathcal{R}.w.x).(s'.3)$, *i.e., the possible value of* v *that* x *can read is* 0. *Hence, we have* $\circledcirc_x(u < v).[1,3].s' = false$ *even though* $\circledcirc(u < v).[1,3].s = true$.

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 *Cmd* below, where $C, C_1, C_2 \in Cmd$.

$$Cmd \;::=\; \mathsf{Magic} \mid \mathsf{Chaos} \mid \mathsf{Idle} \mid [b] \mid v := e \mid C_1\,;\, C_2 \mid C_1 \sqcap C_2 \mid C^{\omega} \mid C_1 \| C_2$$

Thus, a command may either be Magic (an infeasible statement), Chaos (a chaotic statement), Idle (an idle statement), a guard $[b]$, an assignment $v := e$, a sequential composition $(C_1 \,;\, C_2)$, a non-deterministic choice $C_1 \sqcap C_2$, an iteration C^ω, or a parallel composition $C_1 \parallel C_2$.

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 $beh_X.C$, which is defined inductively. For a process x, we let beh_x denote $beh_{\{x\}}$.

$$beh_X.\mathsf{Magic} \;\widehat{=}\; \mathit{false} \tag{3}$$

$$beh_X.\mathsf{Chaos} \;\widehat{=}\; \mathit{true} \tag{4}$$

$$beh_X.\mathsf{Idle} \;\widehat{=}\; \forall x{:}X, v{:}\mathit{Var} \bullet \boxdot \neg \mathcal{W}.v.x \tag{5}$$

$$beh_x.[b] \;\widehat{=}\; \circledast_x b \wedge (\forall v{:}\mathit{Var} \bullet \boxdot \neg \mathcal{W}.v.x) \tag{6}$$

$$beh_x.(v := e) \;\widehat{=}\; \exists k{:}\mathit{Val} \bullet beh_x.[e = k] \,;\, \begin{pmatrix} \boxdot(v = k) \wedge \boxdot \mathcal{W}.v.x \wedge \\ (\forall u{:}\mathit{Var} \backslash \{v\} \bullet \boxdot \neg \mathcal{W}.u.x) \wedge \neg\mathsf{Empty} \end{pmatrix} \tag{7}$$

$$beh_X.(C_1 \,;\, C_2) \;\widehat{=}\; beh_X.C_1 \,;\, beh_X.C_2 \tag{8}$$

$$beh_X.(C_1 \sqcap C_2) \;\widehat{=}\; beh_X.C_1 \vee beh_X.C_2 \tag{9}$$

$$beh_X.C^\omega \;\widehat{=}\; (beh_X.C)^\omega \tag{10}$$

$$beh_X.(C_1 \parallel C_2) \;\widehat{=}\; \begin{array}{l} \exists X_1, X_2 \bullet (X_1 \cup X_2 = X) \wedge (X_1 \cap X_2 = \varnothing) \wedge \\ \qquad beh_{X_1}.(C_1 \,;\, \mathsf{Idle}) \wedge beh_{X_2}.(C_2 \,;\, \mathsf{Idle}) \end{array} \tag{11}$$

We interpret the behaviours of these commands as follows. Note that the behaviours of a guard evaluation (6) and an assignment (7) are only defined for a singleton set $\{x\}$. By (3), command Magic has no behaviours and by (4), Chaos allows any behaviour. By (5), an Idle command executed by set of processes X guarantees that each $x \in X$ does not modify any of the variables of the system. Note that a stronger specification of the behaviour of Idle is $\forall x{:}X, v{:}\mathit{Var} \bullet \boxdot \mathcal{D}.v.x$, which guarantees that each process $x \in X$ is denied access to v. However, we choose to define as weak a specification as possible and allow an implementation of Idle executed by process x to read from a variable v. By (6), 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 (7), 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 write permission to any of the other variables while v is being updated. For any state predicate c, because $\boxdot c$ holds trivially in an empty interval, we must explicitly include conjunct $\neg$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 (8), 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 (9), the behaviour of the non-deterministic choice is the behaviour of either C_1 or C_2. By (10), the behaviour of an iteration C^ω is the behaviour of C iterated a potentially infinite number of times. By (11), 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.

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 $beh_X.C \Rrightarrow beh_X.A$ holds.

We write $A \sqsupseteq_X C$ 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 $C \sqsubseteq_X \mathsf{Magic}$ and $\mathsf{Chaos} \sqsubseteq_X C$ hold trivially.[3]

A convenient approach to constraining the behaviour of a command is to use *enforced conditions*[4], which have been used to derive concurrent programs [Don09, DH12c]. Provided that C is a command and d is an interval predicate, C with an enforced condition d (denoted $\textsc{Enf}\ d \bullet C$) is a command that behaves as C and in addition ensures that d holds [Don09, DH12c]. In particular, the behaviour of $\textsc{Enf}\ d \bullet C$ is defined as follows.

$$beh_X.(\textsc{Enf}\ d \bullet C) \quad \widehat{=} \quad d \wedge beh_X.C$$

Note if $\neg d$ holds, then $\textsc{Enf}\ d \bullet C$ 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.*

$$C \quad \sqsubseteq_X \quad \textsc{Enf}\ d \bullet C \tag{12}$$

$$d' \Rrightarrow d \quad \Rightarrow \quad \textsc{Enf}\ d \bullet C \sqsubseteq_X \textsc{Enf}\ d' \bullet C \tag{13}$$

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 $\circledast b$ and hence every apparent state satisfies b.

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

$$beh_x.(\textsc{Enf}\,(\forall v{:}\, vars.b \bullet \boxdot \neg \mathcal{I}.v.x) \bullet [b]) \quad \Rrightarrow \quad \circledast_x b$$

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.

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

$$
\begin{array}{ll}
 & beh_{\{x,y\}}.((v := v+1) \,||\, (v := v+10)) \\
\Rrightarrow & \text{expanding definitions} \\
 & \exists k_x, k_y \bullet (beh_x.[k_x = v+1]\,;\, (\boxdot(v = k_x) \wedge \boxdot\mathcal{W}.v.x \wedge \neg\mathsf{Empty})\,;\, \boxdot\neg\mathcal{W}.v.x) \wedge \\
 & \qquad (beh_y.[k_y = v+10]\,;\, (\boxdot(v = k_y) \wedge \boxdot\mathcal{W}.v.y \wedge \neg\mathsf{Empty})\,;\, \boxdot\neg\mathcal{W}.v.y)
\end{array}
$$

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

$$
\begin{array}{lcl@{\qquad}lcl}
x_1 & \widehat{=} & beh_x.[k_x = v+1] & y_1 & \widehat{=} & beh_y.[k_y = v+10] \\
x_2 & \widehat{=} & \boxdot(v = k_x) \wedge \boxdot\mathcal{W}.v.x \wedge \neg\mathsf{Empty} & y_2 & \widehat{=} & \boxdot(v = k_y) \wedge \boxdot\mathcal{W}.v.y \wedge \neg\mathsf{Empty} \\
x_3 & \widehat{=} & \boxdot\neg\mathcal{W}.v.x & y_3 & \widehat{=} & \boxdot\neg\mathcal{W}.v.y
\end{array}
$$

Hence, we obtain:

$$
\begin{array}{ll}
 & \exists k_x, k_y \bullet (x_1\,;\, x_2\,;\, x_3) \wedge (y_1\,;\, y_2\,;\, y_3) \\
\equiv & \text{using the write permission to order behaviours} \\
 & \exists k_x, k_y \bullet ((x_1 \wedge (y_1\,;\, y_2\,;\, y_3))\,;\, (x_2 \wedge y_3)\,;\, (x_3 \wedge y_3)) \vee ((y_1 \wedge (x_1\,;\, x_2\,;\, x_3))\,;\, (x_3 \wedge y_2)\,;\, (x_3 \wedge y_3))
\end{array}
$$

Let us consider the first disjunct in more detail.

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

$$(x_1 \wedge (y_1\,;\, y_2))\,;\, (x_2 \wedge y_3)\,;\, (x_3 \wedge y_3)$$

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 $\boxdot c$ holds (for any state predicate c) in an empty interval.

Second, we note that the guard evaluation $beh_x.[k_x = v+1]$ 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 (with **HC2**), 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 $\boxdot\neg\mathcal{W}.v.y$. 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:

$$\exists k_x \bullet \left(x_1 \wedge \left(\begin{array}{l} \boxdot(v = 0 \wedge \neg\mathcal{W}.v.y)\,; \\ (\boxdot(v = 10) \wedge \boxdot\mathcal{W}.v.y \wedge \neg\mathsf{Empty})\,; \\ \boxdot(v = 10 \wedge \neg\mathcal{W}.v.y) \end{array} \right) \right)\,;\, (\boxdot(v = k_x) \wedge \boxdot\mathcal{W}.v.x \wedge \boxdot\neg\mathcal{W}.v.y)\,;\, x_3 \wedge y_3)$$

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:

$$(\boxdot(v = 0)\,;\, \boxdot(v = 10)\,;\, \boxdot(v = 1)) \vee (\boxdot(v = 0)\,;\, \boxdot(v = 10)\,;\, \boxdot(v = 11))$$

Similarly expanding the second disjunct above results in the following behaviour:

$$(\boxdot(v = 0)\,;\ \boxdot(v = 1)\,;\ \boxdot(v = 10)) \vee (\boxdot(v = 0)\,;\ \boxdot(v = 1)\,;\ \boxdot(v = 11))$$

Thus, by allowing execution of 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 during the program's execution, but also the sequence of values that v may have.

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:

$$\begin{array}{ll} & beh_{\{x,y\}}.([u < v] \,\|\, (u := 1\,;\ v := 1)) \\ \Rrightarrow & \text{expanding definitions, using stability properties} \\ & \circledast_x(u < v) \wedge \boxdot(\neg\mathcal{W}.u.x \wedge \neg\mathcal{W}.v.x) \wedge \begin{pmatrix} (\boxdot(u, v = 0, 0) \wedge \boxdot\neg\mathcal{W}.u.y \wedge \boxdot\neg\mathcal{W}.v.y)\,; \\ (\boxdot(u, v = 1, 0) \wedge (\boxdot\mathcal{W}.u.y\,;\ \boxdot\neg\mathcal{W}.u.y))\,; \\ (\boxdot(u, v = 1, 1) \wedge (\boxdot\mathcal{W}.v.y\,;\ \boxdot\neg\mathcal{W}.v.y)) \end{pmatrix} \end{array}$$

Hence, we obtain the scenario described in Example 3, and it is possible for the guard $u < v$ to evaluate to *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 a state predicate (that refers to multiple non-stable variables) holds. 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 is 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, DH12b], 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 $\textsc{Rely}\, r \bullet C$ denote a command with a *rely condition* r, where

$$beh_X.(\textsc{Rely}\, r \bullet C) \;\widehat{=}\; r \Rightarrow beh_X.C$$

Hence, $\textsc{Rely}\, r \bullet C$ 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 $\textsc{Rely}\, r \bullet C$ 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.*

$$r \Rrightarrow r' \;\Rightarrow\; \textsc{Rely}\, r \bullet C \sqsubseteq_X \textsc{Rely}\, r' \bullet C \tag{14}$$
$$r \wedge beh_X.C \Rrightarrow beh_X.A \;\Rightarrow\; \textsc{Rely}\, r \bullet A \sqsubseteq_X \textsc{Rely}\, r \bullet C \tag{15}$$
$$r \wedge beh_X.C \Rrightarrow d \;\Rightarrow\; \textsc{Rely}\, r \bullet \textsc{Enf}\, d \bullet C \sqsubseteq_X \textsc{Rely}\, r \bullet C \tag{16}$$
$$A \sqsubseteq_X \textsc{Enf}\, r \bullet C \;\Leftrightarrow\; \textsc{Rely}\, r \bullet A \sqsubseteq_X C \tag{17}$$

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

Theorem 2 $(\textsc{Rely}\, r \bullet A_1 \| A_2) \sqsubseteq_X C_1 \| C_2$ *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.*

$$(\textsc{Rely}\, r \wedge r_1 \bullet A_1) \sqsubseteq_{X_1} C_1 \;\wedge\; (\textsc{Rely}\, r \wedge r_2 \bullet A_2) \sqsubseteq_{X_2} C_2 \tag{18}$$
$$(r \wedge beh_{X_2}.C_2 \Rrightarrow r_1) \;\wedge\; (r \wedge beh_{X_1}.C_1 \Rrightarrow r_2) \tag{19}$$

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 Δ, 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.*

$$(\textsc{Rely}\, r \bullet C_1 \,;\, C_2) \;\sqsubseteq_X\; (\textsc{Rely}\, r \bullet C_1) \,;\, (\textsc{Rely}\, r \bullet C_2) \tag{20}$$
$$\textsc{Rely}\, r \bullet C^\omega \;\sqsubseteq_X\; (\textsc{Rely}\, r \bullet C)^\omega \tag{21}$$
$$(\textsc{Rely}\, r \bullet A \sqsubseteq_X C) \;\Rightarrow\; (\textsc{Rely}\, r \bullet A^\omega \sqsubseteq_X C^\omega) \tag{22}$$

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, DH12c, DH12b, DH12d]. Furthermore, it is possible to extend this theory to reason about continuous behaviour [DH12c, DH12b, 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 (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 (17) 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

[BCOP05] R. Bornat, C. Calcagno, P. W. O'Hearn, M. J. Parkinson. Permission accounting in separation logic. In Palsberg and Abadi (eds.), *POPL*. Pp. 259–270. ACM, 2005.

[Boy03] J. Boyland. Checking Interference with Fractional Permissions. In Cousot (ed.), *SAS*. LNCS 2694, pp. 55–72. Springer, 2003.

[BW99] R. J. R. Back, J. von Wright. Reasoning algebraically about loops. *Acta Informatica* 36(4):295–334, July 1999.

[CJ07] J. W. Coleman, C. B. Jones. A Structural Proof of the Soundness of Rely/guarantee Rules. *J. Log. Comput.* 17(4):807–841, 2007.

[CZ97] A. Cau, H. Zedan. Refining interval temporal logic specifications. In Bertran and Rus (eds.), *Transformation-Based Reactive Systems Development*. LNCS 1231, pp. 79–94. Springer Berlin / Heidelberg, 1997.

[DH12a] B. Dongol, I. J. Hayes. Approximating idealised real-time specifications using time bands. In *Automated Verification of Critical Systems 2011*. Electronic Communications of the EASST 46, pp. 1–16. EASST, 2012.

[DH12b] B. Dongol, I. J. Hayes. Deriving Real-Time Action Systems Controllers from Multiscale System Specifications. In Gibbons and Nogueira (eds.), *MPC*. LNCS 7342, pp. 102–131. Springer, 2012.

[DH12c] B. Dongol, I. J. Hayes. Deriving real-time action systems in a sampling logic. *Science of Computer Programming*, 2012. Accepted 17 Oct, 2011.

[DH12d] B. Dongol, I. J. Hayes. Rely/guarantee reasoning for teleo-reactive programs over multiple time bands. In *9th International Conference on Integrated Formal Methods*. LNCS 7321, pp. 39–53. 2012.

[DHMS12] B. Dongol, I. J. Hayes, L. Meinicke, K. Solin. Towards an Algebra for Real-Time Programs. In Kahl and Griffin (eds.), *RAMICS*. Lecture Notes in Computer Science 7560, pp. 50–65. Springer, 2012.

[Don09] B. Dongol. *Progress-based verification and derivation of concurrent programs*. PhD thesis, The University of Queensland, 2009.

[Hay08] I. J. Hayes. Towards reasoning about teleo-reactive programs for robust real-time systems. In *SERENE '08*. Pp. 87–94. ACM, New York, NY, USA, 2008.

[HBDJ11] I. J. Hayes, A. Burns, B. Dongol, C. B. Jones. Comparing Models of Nondeterministic Expression Evaluation. Technical report CS-TR-1273, Newcastle University, 2011.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM* 12(10):576–580, 1969.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Prog. Lang. and Syst.* 5(4):596–619, 1983.

[JP11] C. B. Jones, K. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Aspects of Computing* 23:289–306, 2011.

[Mos00] B. C. Moszkowski. A Complete Axiomatization of Interval Temporal Logic with Infinite Time. In *LICS*. Pp. 241–252. 2000.

[OG76] S. Owicki, D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM* 19(5):279–285, 1976.

[Rey02] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. Pp. 55–74. IEEE Computer Society, 2002.

[STER11] G. Schellhorn, B. Tofan, G. Ernst, W. Reif. Interleaved Programs and Rely-Guarantee Reasoning with ITL. *TIME* 0:99–106, 2011.

[SVN+11] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, P. Sewell. Relaxed-memory concurrency and verified compilation. In Ball and Sagiv (eds.), *POPL*. Pp. 43–54. ACM, 2011.